A Nearly Optimal Algorithm for the Geodesic Voronoi Diagram of Points in a Simple Polygon

Chih-Hung Liu

Department of Computer Science, ETH Zürich, Zürich, Switzerland
chih-hung.liu@inf.ethz.ch
https://orcid.org/0000-0001-9683-5982

─── **Abstract** ───

The *geodesic Voronoi diagram* of m point sites inside a simple polygon of n vertices is a subdivision of the polygon into m cells, one to each site, such that all points in a cell share the same nearest site under the geodesic distance. The best known lower bound for the construction time is $\Omega(n + m \log m)$, and a matching upper bound is a long-standing open question. The state-of-the-art construction algorithms achieve $O\big((n+m)\log(n+m)\big)$ and $O(n+m \log m \log^2 n)$ time, which are optimal for $m = \Omega(n)$ and $m = O(\frac{n}{\log^3 n})$, respectively. In this paper, we give a construction algorithm with $O\big(n + m(\log m + \log^2 n)\big)$ time, and it is *nearly optimal* in the sense that if a single Voronoi vertex can be computed in $O(\log n)$ time, then the construction time will become the optimal $O(n + m \log m)$. In other words, we reduce the problem of constructing the diagram in the optimal time to the problem of computing a single Voronoi vertex in $O(\log n)$ time.

1 Introduction

The *geodesic Voronoi diagram* of m point sites inside a simple polygon of n vertices is a subdivision of the polygon into m *cells*, one to each site, such that all points in a cell share the same *nearest site* where the distance between two points is the length of the shortest path between them inside the polygon. The common boundary between two cells is a *Voronoi edge*, and the endpoints of a Voronoi edge are *Voronoi vertices*. A cell can be augmented into *subcells* such that all points in a subcell share the same *anchor*, where the anchor of a point in the cell is the vertex of the shortest path from the associated site to the point that immediately precedes the point. An anchor is either a point site or a *reflex* polygon vertex. Figure 1(a) illustrates an augmented diagram.

The size of the (augmented) diagram is $\Theta(n + m)$ [1]. The best known construction time is $O\big((n+m)\log(n+m)\big)$ [10] and $O(n + m \log m \log^2 n)$ [9]. They are optimal for $m = \Omega(n)$ and for $m = O(\frac{n}{\log^3 n})$, respectively, since the best known lower bound is $\Omega(n + m \log m)$. The existence of a matching upper bound is a long-standing open question by Mitchell [8].

Aronov [1] first proved fundamental properties: a bisector between two sites is a simple curve consisting of $\Theta(n)$ straight and hyperbolic arcs and ending on the polygon boundary; the diagram has $\Theta(n+m)$ vertices, $\Theta(m)$ of which are Voronoi vertices. Then, he developed a divide-and-conquer algorithm that recursively partitions the polygon into two roughly equal-size sub-polygons. Since each recursion level takes $O\big((n+m)\log(n+m)\big)$ time to extend the diagrams between every pair of sub-polygons, the total time is $O\big((n+m)\log(n+m)\log n\big)$.
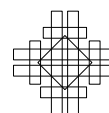
Papadopoulou and Lee [10] combined the divide-and-conquer and plane-sweep paradigms to improve the construction time to $O\big((n+m)\log(n+m)\big)$. First, the polygon is triangulated and one resultant triangle is selected as the root such that the dual graph is a rooted binary tree and each pair of adjacent triangles have a parent-child relation; see Figure 1(b). For each triangle, its diagonal shared with its parent partitions the polygon into two sub-polygons: the "*lower*" one contains it, and the "*upper*" one contains its parent. Then, the triangles are swept by the post-order and pre-order traversals of the rooted tree to respectively build, inside each triangle, the two diagrams with respect to sites in its lower and upper sub-polygons. Finally, the two diagrams inside each triangle are merged into the final diagram.

Very recently, Oh and Ahn [9] generalized the notion of plane sweep to a simple polygon. To supplant the scan line, one point is fixed on the polygon boundary, and another point is moved from the fixed point along the polygon boundary counterclockwise, so that the shortest path between the two points will sweep the whole polygon. Moreover, Guibas and Hershberger's data structure for shortest path queries [4, 6] is extended to compute a Voronoi vertex among three sites or between two sites in $O(\log^2 n)$ time. This technique enables handling an event in $O(\log m \log^2 n)$ time, leading to a total time of $O(n + m \log m \log^2 n)$.

Papadopoulou and Lee's method [10] has two issues inducing the $n \log(n+m)$ time-factor. First, while sweeping the polygon, an intermediate diagram is represented by a "*wavefront*" in which a "*wavelet*" is associated with a "subcell." Although this representation enables computing the common vertex among three subcells in $O(1)$ time, since it takes $\Omega\big(\log(n+m)\big)$ time to update such a wavefront, the $\Omega(n)$ vertices lead to the $n \log(n + m)$ factor. Second, when a wavefront enters a triangle from one diagonal and leaves from the other two diagonals, it will split into two. Since there are $\Omega(n)$ triangles, there are $\Omega(n)$ split events, and since a split event takes $\Omega\big(\log(n + m)\big)$ time, the $n \log(n + m)$ factor arises again.

1.1   Our contribution

We devise a construction algorithm with $O\big(n + m(\log m + \log^2 n)\big)$ time, which is slightly faster than Oh and Ahn's method [9] and is optimal for $m = O(\frac{n}{\log^2 n})$. More importantly, our algorithm is, to some extent, *nearly optimal* since the $\log^2 n$ factor solely comes from computing a single Voronoi vertex. If the computation time can be improved to $O(\log n)$, the total construction time will become $O\big(n + m(\log m + \log n)\big)$, which equals the optimal $O(n + m \log m)$ since $m \log n = O(n)$ for $m = O(\frac{n}{\log n})$ and $\log n = O(\log m)$ for $m = \Omega(\frac{n}{\log n})$. In other words, we reduce the problem of constructing the diagram in the optimal time by Mitchell [8] to the problem of computing a single Voronoi vertex in $O(\log n)$ time.

At a high level, our algorithm is a new implementation of Papadopoulou and Lee's concept [10] using a different data structure of a wavefront, symbolic maintenance of incomplete Voronoi edges, tailor-made wavefront operations, and appropriate amortized time analysis.

First, in our wavefront, each wavelet is directly associated with a cell rather than a subcell. This representation makes use of Oh and Ahn's [9] $O(\log^2 n)$-time technique of computing a Voronoi vertex. Each wavelet also stores the anchors of incomplete subcells in its associated cell in order to enable locating a point in a subcell along the wavefront.

Second, if each change of a wavefront needs to be updated immediately, a priority queue for events would be necessary, and since the diagram has $\Theta(m+n)$ vertices, an $(m+n)\log(m+n)$ time-factor would be inevitable. To overcome this issue, we maintain incomplete Voronoi edges symbolically, and update them only when necessary. For example, during a binary search along a wavefront, each incomplete Voronoi edge of a tested wavelet will be updated.

Third, to avoid $\Omega(n)$ split operations, we design two tailor-made operations. If a wavefront will separate into two but one part will not be involved in the follow-up "sweep", then, instead of using a binary search, we "divide" the wavefront in a seemingly brute-force way in which

we traverse the wavefront from the uninvolved part until the "division" point, remove all visited subcells, and build another wavefront from those subcells. If a wavefront propagates into a sub-polygon that contains no point site, then we adopt a two-phase process to build the diagram inside the sub-polygon instead of splitting a wavefront many times.

Finally, when deleting or inserting a subcell (anchor), its position in a wavelet is known. Since re-balancing a *red-black tree* (RB-tree) after an insertion or a deletion takes amortized $O(1)$ time [7, 11, 12], by augmenting each tree node with pointers to its predecessor and successor, an insertion or a deletion with a known position takes amortized $O(1)$ time.

This paper is organized as follows. Section 2 formulates the geodesic Voronoi diagram, defines a rooted partition tree, and introduces Papadopoulou and Lee's two subdivisions [10]; Section 3 summarizes our algorithm; Section 4 designs the data structure of a wavefront; Section 5 presents wavefront operations; Section 6 implements the algorithm with those operations. Due to the page limit, we omit some technical details and proofs; for more details, please see the full version.

2 Preliminary
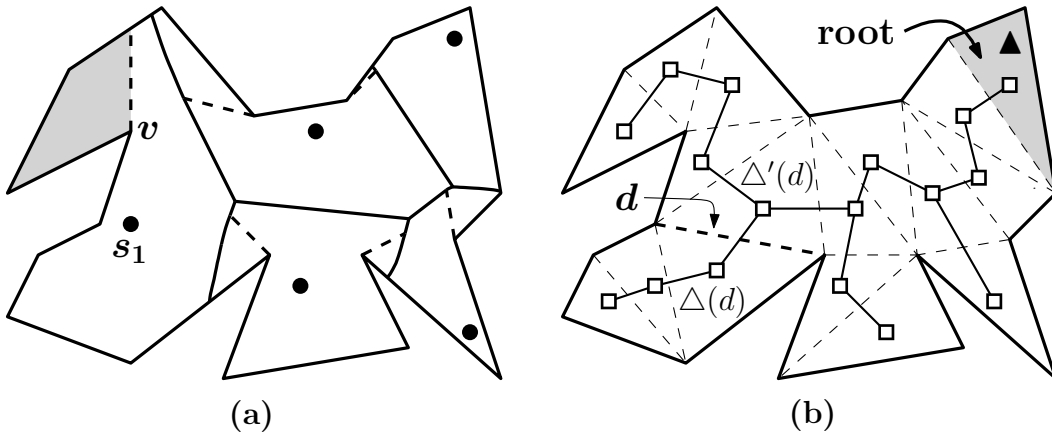
2.1 Geodesic Voronoi diagrams

Let P be a simple polygon of n vertices, let ∂P denote the boundary of P, and let S be a set of m point sites inside P. For any two points p, q in P, the *geodesic distance* between them, denoted by $d(p, q)$, is the length of the shortest path between them that fully lies in P, the *anchor* of q with respect to p is the last vertex on the shortest path from p to q before q, and the *shortest path map* (SPM) from p in P is a subdivision of P such that all points in a region share the same anchor with respect to p. Each edge in the SPM from p is a line segment from a reflex polygon vertex v of P to ∂P along the direction from the anchor of v (with respect to p) to v, and this line segment is called the *SPM edge* of v (from p).

The *geodesic Voronoi diagram* of S in P, denoted by $\mathrm{Vor}_P(S)$, partitions P into m *cells*, one to each site, such that all points in a cell share the same nearest site in S under the geodesic distance. The cell of a site s can be augmented by partitioning the cell with the SPM from s into *subcells* such that all points in a subcell share the same anchor with respect to s. The augmented version of $\mathrm{Vor}_P(S)$ is denoted by $\mathrm{Vor}_P^*(S)$. With a slight abuse of terminology, a cell in $\mathrm{Vor}_P^*(S)$ indicates a cell in $\mathrm{Vor}_P(S)$ together with its subcells in $\mathrm{Vor}_P^*(S)$. Then, each cell is associated with a site, and each subcell is associated with an anchor. As shown in Figure 1(a), v is the anchor of the shaded subcell (in s_1's cell), and the last vertex on the shortest path from s_1 to any point x in the shaded subcell before x is v.

A *Voronoi edge* is the common boundary between two adjacent cells, and the endpoints of a Voronoi edge are called *Voronoi vertices*. A Voronoi edge is a part of the *geodesic bisector* between the two associated sites, and consists of straight and hyperbolic arcs. Endpoints of these arcs except Voronoi vertices are called *breakpoints*, and a breakpoint is incident to an SPM edge in the SPM from one of the two associated sites, indicating a change of the corresponding anchor. There are $\Theta(m)$ Voronoi vertices and $\Theta(n)$ breakpoints [1].

In our algorithm, each anchor u refers to either a reflex polygon vertex of P or a point site in S; we store its associated site s, its geodesic distance from s, and its anchor with respect to s. The weighted distance from u to a point x is $d(s, u) + |\overline{ux}|$.

Throughout the paper, we make a general position assumption that no polygon vertex is equidistant from two sites in S and no point inside P is equidistant from four sites in S. The former avoids nontrivial overlapping among cells [1], and the latter ensures that the degree of each Voronoi vertex with respect to Voronoi edges is either 1 (on ∂P) or 3 (among three cells).

Figure 1 (a) Augmented geodesic Voronoi diagram $\mathrm{Vor}_P^*(S)$. (b) Rooted partition tree $\mathcal{T}$.

The boundary of a cell except Voronoi edges are polygonal chains on ∂P. For convenience, these polygonal chains are referred to as *polygonal edges* of the cell, the incidence of an SPM edge onto a polygonal edge is also a *breakpoint*, and the polygonal edges including their polygon vertices and breakpoints also count for the size of the cell.

▶ **Lemma 1.** *([9, Lemma 5 and 14]) It takes $O(\log^2 n)$ time to compute the degree-1 or degree-3 Voronoi vertex between two sites or among three sites after $O(n)$-time preprocessing.*

2.2   A rooted partition tree

Following Papadopoulou and Lee [10], a rooted partition tree $\mathcal{T}$ for P and S is built as follows: First, P is triangulated using Chazelle's algorithm [2] in $O(n)$ time, and all sites in S are located in the resulting triangles by Edelsbrunner et al's approach [3] in $O(n + m \log n)$ time. The dual graph of the triangulation is a tree in which each node corresponds to a triangle and an edge connects two nodes if and only if their corresponding triangles share a diagonal. Then, an arbitrary triangle ▲ whose two diagonals are polygon sides, i.e., a node with degree 1, is selected as the *root*, so that there is a parent-child relation between each pair of adjacent triangles. Figure 1(b) illustrates a rooted partition tree $\mathcal{T}$.

For a diagonal d, let $\triangle(d)$ and $\triangle'(d)$ be the two triangles adjacent to d such that $\triangle'(d)$ is the parent of $\triangle(d)$, and call d the *root diagonal* of $\triangle(d)$; also see Figure 1(b). d partitions P into two sub-polygons: $P(d)$ contains $\triangle(d)$ and $P'(d)$ contains $\triangle'(d)$. $P(d)$ and $P'(d)$ are said to be "below" and "above" d, respectively. Assume that $d \subseteq P(d)$; let $S(d) = S \cap P(d)$ and $S'(d) = S \setminus S(d)$, which indicate the two respective subsets of sites below and above d.

In this paper, we adopt the following convention: for each triangle $\triangle$, let $d = \overline{v_1 v_2}$ be its root diagonal, let $\triangle'$ be its parent triangle, let d_1, d_2 be the other two diagonals of $\triangle$, and let d_3, d_4 be the other two diagonals of $\triangle'$. Assume that d_4 is the diagonal between $\triangle'$ and its parent, and that d_1, d, and d_4 are incident to v_1, and d_2, d, and d_3 are incident to v_2. Let $v_{1,2}$ be the vertex shared by d_1 and d_2, and let $v_{3,4}$ be the vertex shared by d_3 and d_4. Denote the set of sites in $\triangle$ as $S_\triangle = S(d) - S(d_1) - S(d_2)$. Figure 2(a) shows an illustration.

2.3   Subdivisions

Papadopoulou and Lee [10] introduced two subdivisions, $\mathcal{SD}$ and $\mathcal{SD}'$, of P, which can be merged into $\mathrm{Vor}_P^*(S)$. For each triangle $\triangle$ with a root diagonal d, $\mathcal{SD}$ and $\mathcal{SD}'$ respectively contain $\mathrm{Vor}_P^*\big(S(d)\big) \cap \triangle$ and $\mathrm{Vor}_P^*\big(S'(d)\big) \cap \triangle$; $\mathcal{SD}$ also contains $\mathrm{Vor}_P^*(S) \cap$ ▲. Since $S(d)$
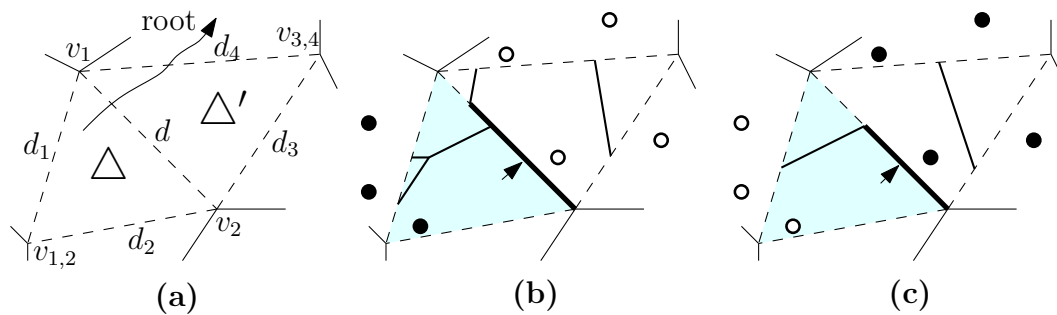
Figure 2 (a) $\triangle$ and $\triangle'$. (b) $\mathcal{SD} \cap \triangle$. (c) $\mathcal{SD}' \cap \triangle$. (Borders on d are indicated by arrows.)

and $S(d_4)$ (resp. $S'(d)$ and $S'(d_4)$) may differ, a *border* forms along d in $\mathcal{SD}$ (resp. $\mathcal{SD}'$) to "remedy" the conflicting proximity information. Figure 2(b)–(c) illustrate $\mathcal{SD}$ and $\mathcal{SD}'$.

The incidence of a Voronoi edge or an SPM edge in $\mathrm{Vor}_P^*(S)$ onto a border in $\mathcal{SD}$ or $\mathcal{SD}'$ is called a *border vertex*, and the border vertices partition a border into *border edges*. Both $\mathcal{SD}$ and $\mathcal{SD}'$ have $O(n+m)$ border vertices [10], $O(m)$ of which are induced by Voronoi edges. Hereafter, a *diagram vertex* means a Voronoi vertex, a breakpoint, a border vertex, or a polygon vertex.

3 Overview of the algorithm

We compute $\mathrm{Vor}_P^*(S)$ in the following three steps:
1. Build the rooted partition tree $\mathcal{T}$ for P and S in $O(n + m \log n)$ time. (Section 2.2)
2. Construct $\mathcal{SD}$ and $\mathcal{SD}'$ in $O\big(n + m(\log m + \log^2 n)\big)$ time by sweeping the polygon using the post-order and pre-order traversals of $\mathcal{T}$, respectively. (Section 6.1 and Section 6.2)
3. Merge $\mathcal{SD}$ and $\mathcal{SD}'$ into $\mathrm{Vor}_P^*(S)$ in $O(n + m)$ time using Papadopoulou and Lee's method [10, Section 7].

By the above-mentioned running times, we conclude the total running time as follows.

▶ **Theorem 2.** $\mathrm{Vor}_P^*(S)$ *can be constructed in* $O\big(n + m(\log m + \log^2 n)\big)$ *time.*

4 Wavefront structure

A *wavefront* represents the "*incomplete*" boundary of "*incomplete*" Voronoi cells during the execution of our algorithm, and wavefronts will "*sweep*" the simple polygon P triangle by triangle to construct $\mathcal{SD}$ and $\mathcal{SD}'$. To avoid excessive updates, each "*incomplete*" Voronoi edge, which is a part of a Voronoi edge and will be completed during the sweep, is maintained *symbolically*, preventing an extra $\log n$ time-factor. During the sweep, candidates for Voronoi vertices in $\mathcal{SD}$ and $\mathcal{SD}'$ called *potential vertices* will be generated in the *unswept* part of P.

4.1 Formal definition and data structure

Let η be a diagonal or a pair of diagonals sharing a common polygon vertex, and let S' be a subset of S lying on the same side of η. A *wavefront* $W_\eta(S')$ represents the sequence of Voronoi cells in $\mathrm{Vor}_P^*(S')$ appearing along η, and each appearance of a cell induces a *wavelet* in $W_\eta(S')$. The *unswept* area of $W_\eta(S')$ is the part of P on the opposite side of η from S'. Since $\mathrm{Vor}_P^*(S')$ in the unswept area has not yet been constructed, Voronoi and polygonal

edges incident to η are called *incomplete.* Each wavelet is bounded by two *incomplete* Voronoi or polygonal edges along η, and its *incomplete boundary* comprises its two incomplete edges and the portion of η between them. When the context is clear, a wavelet may indicate its associated cell.

$W_\eta(S')$ is stored in an RB-tree in which each node refers to one wavelet and the ordering of nodes follows their appearances along η. The RB-tree is augmented such that each node has pointers to its predecessor and successor, and the root has pointers to the first and last nodes, enabling efficiently traversing wavelets along η and accessing the two ending wavelets.

The subcells of a wavelet are the subcells in its associated cell incident to its incomplete boundary. The list of their anchors is also maintained by an augmented RB-tree in which their ordering follows their appearances along the incomplete boundary. Due to the visibility of a subcell, each subcell appears exactly once along the incomplete boundary. Since the rebalancing after an insertion or a deletion takes amortized $O(1)$ time [11, 12, 7], inserting or deleting an anchor at a known position in an RB-tree, i.e., without searching, takes amortized $O(1)$ time.

4.2   Incomplete Voronoi and polygonal edges

When a wavefront moves into its *unswept* area, incomplete Voronoi edges will extend, generating new breakpoints. If each breakpoint needs to be created immediately, all candidates for breakpoints should be maintained in a priority queue, leading to an $\Omega(n \log n)$ running time due to $\Omega(n)$ breakpoints. To avoid these excessive updates, we maintain each incomplete Voronoi edge *symbolically*, and update it only when necessary. For example, when searching along the wavefront or merging two wavefronts, the incomplete Voronoi edges of each involved wavelet (cell) will be updated until the diagonal or the pair of diagonals.

Since a breakpoint indicates the change of a corresponding anchor, for a Voronoi edge, if the anchors of its incident subcells on "each side" are stored in a sequence, the Voronoi edge can be computed in time proportional to the number of breakpoints by scanning the two sequences [9, Section 4]. Following this concept, for each incomplete Voronoi edge, we store its fixed Voronoi vertex (in the swept area), its last created breakpoint, and its last used anchors on its two sides, so that we can update an incomplete Voronoi edge by scanning the two lists of anchors from the last used anchors. When creating a breakpoint, we also build a corresponding SPM edge, and then remove the corresponding anchor from the anchor list.

Each polygonal edge is also maintained symbolically in a similar way; in particular, it will also be updated when a polygon vertex is inserted as an anchor. Meanwhile, the SPM edges incident to a polygonal edge will also be created using its corresponding anchor list.

4.3   Potential vertices

We process incomplete Voronoi edges to generate candidates for Voronoi vertices called *potential vertices.* For each incomplete Voronoi edge, since its two associated sites lie in the swept area, one endpoint of the corresponding bisector lies in the unswept area and is a degree-1 potential vertex. For each two adjacent Voronoi edges along the wavefront, their respective bisectors may intersect in the unswept area, and the intersection is a degree-3 potential vertex. By Lemma 1, a potential vertex can be computed in $O(\log^2 n)$ time.

Potential vertices are stored in their located triangles; each diagonal of a triangle is equipped with a priority queue to store the potential vertices associated with sites in the triangles on the opposite side of the diagonal, where the key is the distance to the diagonal.

5 Wavefront operations

We summarize the eight wavefront operations, where K_{inv}, A_{vis} and I_{new} are the numbers of involved (i.e., processed and newly created) potential vertices, visited anchors, and created diagram vertices, respectively:

Initiate: Compute $\text{Vor}_\triangle(S_\triangle)$ and initialize $W_{(d,d_2)}(S_\triangle)$ in $O\big(|S_\triangle|(\log m + \log^2 n)\big)$ time.

Extend: Extend one wavefront into a triangle from one diagonal to the opposite pair of diagonals to build the diagram inside the triangle in $O\big(K_{\text{inv}}(\log m + \log^2 n) + I_{\text{new}}\big)$ plus amortized $O(1)$ time.

Merge: Merge two wavefronts sharing the same diagonal or the same pair of diagonals together with merging the two corresponding diagrams in $O\big(|S_\triangle|(\log m + \log n) + K_{\text{inv}}(\log m + \log^2 n) + I_{\text{new}}\big)$ plus amortized $O(1)$ time where $\triangle$ is the underlying triangle.

Join: Join two wavefronts sharing the same diagonal with building the border on the diagonal but without merging the two corresponding diagrams inside the underlying triangle $\triangle'$ in $O\big(|S_{\triangle'}|(\log m + \log n) + K_{\text{inv}}(\log m + \log^2 n) + I_{\text{new}}\big)$ plus amortized $O(1)$ time.

Split: Split a wavefront using a binary search in $O(\log m + \log n)$ time.

Divide: Divide a wavefront by traversing one diagonal in amortized $O(A_{\text{vis}} + 1)$ time.

Insert: Insert $S_\triangle$ into $W_{d_1}\big(S(d_2)\big)$ to be $W_{d_1}\big(S(d_2) \cup S_\triangle\big)$ in $O\big(|S_\triangle|(\log m + \log^2 n)\big)$ time.

Propagate: Propagate $W_d\big(S'(d)\big)$ into $P(d)$, provided that $P(d) \cap S = S(d) = \emptyset$, to build $\mathcal{SD}' \cap P(d) = \text{Vor}_P^*(S) \cap P(d)$ in $O\big(K_{\text{inv}}(\log m + \log^2 n) + |\mathcal{SD}' \cap P(d)|\big)$ time.

Merge and Join operations differ in that the former also merges the two corresponding Voronoi diagrams inside the underlying triangle, and the latter does not.

Readers could directly read the algorithm in Section 6 without knowing the detailed implementation for wavefront operations. For the operation times, we have three remarks.

▶ Remark. During Extend and Propagate operations and at the end of the other operations, potential vertices will be generated according to new incomplete Voronoi edges. It will be clear in Section 6 that the total number of potential vertices in the construction of $\mathcal{SD}$ and $\mathcal{SD}'$ is $O(m)$, so a priority queue takes $O(\log m)$ time for an insertion or an extraction.

▶ Remark. As stated in Section 4.2, we maintain incomplete Voronoi/polygonal edges symbolically. For the sake of simplicity, we charge the time to update an incomplete edge to the created breakpoints, and assign the corresponding costs to their located triangles.

▶ Remark. Since a wavelet (resp. anchor) to remove from a wavefront (resp. wavelet) must be inserted beforehand, we charge its removal cost at its insertion time. For a wavelet, the cost is $O(\log m)$, and for an anchor, since the position is always known, the cost is amortized $O(1)$. Similarly, we charge the cost to delete a diagram vertex at its creation time.

Due to the page limit, we omit Initiate, Split, Insert, and Join operations. The first three are quite straightforward, and the last one is similar to a Merge operation; for more details, please see the full version.

5.1 Extend operation

An Extend operation extends a wavefront $W_{\tilde{d}}(Q)$ from one diagonal $\tilde{d}$ of a triangle $\tilde{\triangle}$ to the opposite pair of diagonals $(\tilde{d}_1, \tilde{d}_2)$ to construct $W_{(\tilde{d}_1, \tilde{d}_2)}(Q)$ and $\text{Vor}_P^*(Q) \cap \tilde{\triangle}$, where $\tilde{d} = \overline{\tilde{v}_1 \tilde{v}_2}$, $\tilde{d}_1 = \overline{\tilde{v}_1 \tilde{v}_{1,2}}$, and $\tilde{d}_2 = \overline{\tilde{v}_2 \tilde{v}_{1,2}}$, and Q lies on the opposite side of $\tilde{d}$ from $\tilde{\triangle}$.

This operation is equivalent to sweeping the triangle with a scan line parallel to $\tilde{d}$ and processing each hit potential vertex. The next hit potential vertex is provided from the priority queue associated with $\tilde{d}$ (defined in Section 4.3), and will be processed in three phases: First, its validity is verified: for a degree-1 potential vertex, its incomplete Voronoi

edge should be alive in the wavefront, and for a degree-3 one, its two incomplete Voronoi edges should be still adjacent in the wavefront. Second, the one or two incomplete Voronoi edges are updated up to the potential vertex. Since a degree-1 potential vertex is incident to a polygonal edge, the polygonal edge is also updated.

Finally, when a potential vertex becomes a Voronoi vertex, a wavelet will be removed from the wavefront. For a degree-1 potential vertex, since the wavelet lies at one end of the wavefront, a polygonal edge is added to its adjacent wavelet; for a degree-3 potential vertex, since the removal makes two wavelets adjacent, a new incomplete Voronoi edge is created, and one degree-1 and two degree-3 potential vertices are computed accordingly.

After the extension, if $\tilde{v}_{1,2}$ is a reflex polygon vertex, the wavefront is further processed by three cases. If neither $\tilde{d}_1$ nor $\tilde{d}_2$ is a polygon side, $\tilde{v}_{1,2}$ will later be inserted as an anchor while dividing or splitting $W_{(\tilde{d}_1, \tilde{d}_2)}(Q)$ at $\tilde{v}_{1,2}$. If both $\tilde{d}_1$ and $\tilde{d}_2$ are polygon sides, all the subcells will be completed along $(\tilde{d}_1, \tilde{d}_2)$ and the wavefront will disappear. If only $\tilde{d}_1$ (resp. $\tilde{d}_2$) is a polygon side, all the subcells along $\tilde{d}_1$ (resp. $\tilde{d}_2$) excluding the one containing $\tilde{v}_{1,2}$ will be completed, and $\tilde{v}_{1,2}$ will be inserted into the corresponding wavelet as the last or first anchor.

The operation time is $O\big(K_{\mathrm{inv}}(\log m + \log^2 n) + I_{\mathrm{new}}\big)$ plus amortized $O(1)$, where K_{inv} is the number of involved (i.e., processed and newly created) potential vertices and I_{new} is the number of created diagram vertices. First, extracting a potential vertex from the priority queue takes $O(\log m)$ time, and verifying its validity takes $O(1)$ time. Second, updating incomplete Voronoi and polygonal edges takes time linear in the number of created breakpoints (Section 4.2), i.e., $O(I_{\mathrm{new}})$ time in total. Third, since computing a potential vertex takes $O(\log^2 n)$ time, locating it takes $O(\log n)$ time, and inserting it into a priority queue takes $O(\log m)$ time, creating a new potential vertex takes $O(\log^2 n + \log m)$ time. Finally, completing subcells takes $O(I_{\mathrm{new}})$ time, and inserting $\tilde{v}_{1,2}$ takes amortized $O(1)$ time since its position in the anchor list is known.

5.2   Merge operation

A Merge operation merges $W_\eta(Q)$ and $W_\eta(Q')$ into $W_\eta(Q \cup Q')$ together with $\mathrm{Vor}_P^*(Q) \cap \triangle$ and $\mathrm{Vor}_P^*(Q') \cap \triangle$ into $\mathrm{Vor}_P^*(Q \cup Q') \cap \triangle$ where $\triangle$ is the underlying triangle. In our algorithm, either $Q = S_\triangle$, $Q' = S(d_1)$, and $\eta = (d, d_2)$ or $Q = S_\triangle \cup S(d_1)$, $Q' = S(d_2)$, and $\eta = d$; in both cases, $S_\triangle \subseteq Q$. The border will form on $\partial\triangle \setminus \eta$, i.e., d_1 for the former case and (d_1, d_2) for the latter case. Although a wavefront only stores incomplete Voronoi cells, its associated diagram can still be accessed through the Voronoi edges of the stored cells. After the merge, new incomplete Voronoi edges will form, and their potential vertices will be created.

The Merge operation consists of two phases: (1) merge $\mathrm{Vor}_P^*(Q) \cap \triangle$ and $\mathrm{Vor}_P^*(Q') \cap \triangle$ into $\mathrm{Vor}_P^*(Q \cup Q') \cap \triangle$ and (2) merge $W_\eta(Q)$ and $W_\eta(Q')$ into $W_\eta(Q \cup Q')$.

The first phase is to construct so-called *merge curves*. A merge curve is a connected component consisting of border edges along $\partial\triangle \setminus \eta$ and Voronoi edges in $\mathrm{Vor}_P^*(Q \cup Q') \cap \triangle$ associated with one site in Q and one site in Q'; the ordering of merge curves is the ordering of their appearances along η. This phase is almost identical to the merge process by Papadopoulou and Lee [10, Section 5], but since our data structure for a wavefront is different from theirs, a binary search along a wavefront to find a starting endpoint for a merge curve requires a different implementation. Due to the page limit, we only state this difference here; for the details of tracing a merge curve, please see the full version.

Assume η to be oriented from v_1 to $v_{1,2}$ for $\eta = (d, d_2)$ and from v_1 to v_2 for $\eta = d$. By [10, Lemma 4–6], a merge curve called *initial* starts from $v_{1,2}$ for the latter case ($\eta = d$), but all other merge curves have both endpoints on η, and those endpoints are associated with one site in $S_\triangle$. Let $Q_\triangle$ be the set of sites in $S_\triangle$ that have a wavelet in $W_\eta(Q)$. If $\eta = (d, d_2)$, a

site in $Q_\triangle$ can have two wavelets in $W_\eta(Q)$, and with an abuse of terminology, such a site is imagined to have two copies, each to one wavelet. Since $Q_\triangle \subseteq Q$, finding a starting endpoint for each merge curve except the initial one is to test sites in $Q_\triangle$ following the ordering of their wavelets in $W_\eta(Q)$ along η. After finding a starting endpoint, the corresponding merge curve will be traced; when the tracing reaches η again, a stopping endpoint forms, and the first site in $Q_\triangle$ lying after the site inducing the stopping endpoint will be tested.

Let x be the next starting endpoint, which is unknown, and let s be the next site in $Q_\triangle$ to test. A two-level binary search on $W_\eta(Q')$ determines if s induces x, and if so, further determines the site $t \in Q'$ that induces x with s as well as the corresponding anchor.

The first-level binary search executes on the RB-tree for the wavelets in $W_\eta(Q')$, and each step determines for a site $q \in Q'$ if its cell lies before or after t's cell along η or if $q = t$. Let y_1 and y_2 denote the two intersection points between η and the two incomplete edges of s (in $W_\eta(Q)$), where y_1 lies before y_2 along η, and let z_1 and z_2 be the two points defined similarly for q (in $W_\eta(Q')$). Since s lies in $\triangle$, the distance between s and any point in η can be computed in $O(1)$ time. The two incomplete edges of q will be updated until z_1 and z_2, so that the distance from q to z_1 (resp. to z_2) can be computed from the corresponding anchor. For example, if u is the anchor of the subcell that contains z_1, $d(z_1, q) = |\overline{z_1 u}| + d(u, q)$.

The determination considers four cases. (Assume s and t induce the "starting" endpoint.)

- z_2 lies before y_1 (resp. z_1 lies after y_2): t's cell lies after (resp. before) q's cell.
- z_1 lies before y_1 and z_2 lies between y_1 and y_2 (resp. z_2 lies after y_2 and z_1 lies between y_1 and y_2): if z_2 is closer to q than to s (resp. z_1 is closer to q than to s), then t's cell lies after (resp. before) q's cell; otherwise, t is q.
- Both y_1 and y_2 lie between z_1 and z_2: t is q.
- Both z_1 and z_2 lie between y_1 and y_2: let x_s be the projection point of s onto η.
  - If x_s lies before z_1, then t's cell lies before q's cell.
  - If x_s lies after z_2: if z_2 is closer to q than to s, t's cell lies after q's cell; if both z_1 and z_2 are closer to s than to q, t's cell lies before q's cell; otherwise, $t = q$.
  - If x_s lies between z_1 and z_2: if z_1 is closer to s than to q, t's cell lies before q's cell; otherwise, $t = q$.

If the first-level search does not find t, then s does not induce the next starting endpoint x.

The second-level binary search executes on the RB-tree for t's anchor list to either determine the next starting endpoint x and t's corresponding anchor or report that s does not induce x. Let u be the current anchor of t to test, and let x_s be the projection point of s onto η. u's "interval" on η can be decided by checking u's two neighboring anchors. If u's interval lies after x_s, x lies before u's interval; otherwise, if both endpoints of u's interval are closer to (resp. farther from) t than to (resp. from) s, x lies after (resp. before) u's interval, and if one endpoint is closer to t than to s but the other is not, then x lies in u's interval and can be computed in $O(1)$ time since $d(x, s) = |\overline{xu}| + d(u, t)$. If the second-level binary search does not find such an interval, s does not induce the next starting endpoint x.

The second phase (i.e., merging $W_\eta(Q)$ and $W_\eta(Q')$ into $W_\eta(Q \cup Q')$) splits $W_\eta(Q)$ and $W_\eta(Q')$ at the endpoints of merge curves, and concatenates active parts at these endpoints where a part is called active if it contributes to $W_\eta(Q \cup Q')$. In fact, the active parts along η alternately come from $W_\eta(Q)$ and $W_\eta(Q')$. At each merging endpoint, the two cells become adjacent, generating a new incomplete Voronoi edge. Potential vertices of these incomplete Voronoi edges will be computed and inserted into the corresponding priority queues. For each ending polygon vertex of η, if it is reflex but has not yet been an anchor of $W_\eta(Q \cup Q')$, it will be inserted into its located wavelet as the first or the last anchor.

The total operation time is $O\big(|S_\triangle|(\log n + \log m) + K_{\text{new}}(\log m + \log^2 n) + I_{\text{new}}\big)$ plus amortized $O(1)$, where K_{new} is the number of created potential vertices and I_{new} is the number of created diagram vertices while merging the two diagrams. First, since each two-level binary search takes $O(\log m + \log n)$ time, finding starting points takes $O\big(|S_\triangle|(\log m + \log n)\big)$ time. Second, by [10, Section 5], tracing a merge curve takes time linear in the number of deleted and created vertices, but the time to delete vertices has been charged at their creation, implying that tracing all the merge curves takes $O(I_{\text{new}})$ time.

Third, an incomplete Voronoi edge generates at least one potential vertex, so the number of new incomplete Voronoi edges is $O(K_{\text{new}})$. Since an endpoint of a merge curve corresponds to a new incomplete Voronoi edge, there are $O(K_{\text{new}})$ split and $O(K_{\text{new}})$ concatenation operations, and since each operation takes $O(\log m + \log n)$ time, it takes $O\big(K_{\text{new}}(\log m + \log n)\big)$ time to merge the two wavefronts. By the same analysis in Section 5.1, creating K_{new} potential vertices takes $O\big(K_{\text{new}}(\log m + \log^2 n)\big)$ time. Finally, inserting an ending polygon vertex of η as an anchor takes amortized $O(1)$ time.

5.3   Divide operation

A Divide operation divides a wavefront associated with a pair of diagonals at the common polygon vertex by traversing one diagonal instead of using a binary search. Although a Divide operation seems a brute-force way compared to a Split operation, since a Split operation takes $\Omega(\log n + \log m)$ time and there are $\Omega(n)$ events to separate a wavefront, if only Split operations are adopted, the total construction time would be $\Omega\big(n(\log n + \log m)\big)$.

First, the wavefront is traversed from the end of the selected diagonal subcell by subcell, i.e., anchor by anchor, until reaching the common vertex. Then, the wavefront is separated at the common polygon vertex by removing all the visited anchors except the last one, duplicating the last one, and building a new wavefront for these "removed" anchors and the duplicate anchor from scratch. Finally, if the common polygon vertex is reflex, it is inserted into its located wavelets in both resultant wavefronts as an anchor without a binary search (since it is the first or last anchor of its located wavelets).

The total operation time is amortized $O(A_{\text{vis}} + 1)$, where A_{vis} is the number of visited anchors. Since each wavelet (resp. anchor) records its two neighboring wavelets (resp. anchors) in the augmented RB-tree, the time to locate the common polygon vertex is $O(A_{\text{vis}})$. Recall that a cell must have one subcell, and the time to remove a wavelet or an anchor has been charged when it was inserted. Finally, building the new wavefront from scratch takes amortized $O(A_{\text{vis}})$ time, and inserting the common vertex takes amortized $O(1)$ time.

5.4   Propagate operation

A Propagate operation propagates a wavefront $W_d\big(S'(d)\big)$ into $P(d)$, i.e., from the upside to the downside of d, to build $\mathcal{SD}' \cap P(d)$ provided that $S \cap P(d) = S(d) = \emptyset$. Since $S(d) = \emptyset$, then $\mathcal{SD}' \cap P(d)$ is exactly $\text{Vor}_P^*(S) \cap P(d)$. To some extent, a Propagate operation is a generalized version of an Extend operation underlying a sub-polygon instead of a triangle.

The operation consists of two phases. The first phase constructs $\text{Vor}_P(S) \cap P(d)$, and the second phase refines each cell into subcells to obtain $\text{Vor}_P^*(S) \cap P(d)$.

The first phase "sweeps" the triangles in $P(d)$ by a preorder traversal of the subtree of $\mathcal{T}$ rooted at $\triangle(d)$. Similar to the Extend operation, this sweep processes potential vertices inside each triangle to construct Voronoi edges and to update the wavefront accordingly. However, this sweep will not process the polygon vertices of $P(d)$, so that the anchor lists will not be updated, preventing constructing a Voronoi edge using the two corresponding

anchor lists. Fortunately, Oh and Ahn [9] gave another technique that obtains in $O(\log n)$ time the two anchor lists of a Voronoi edge provided that the two Voronoi vertices are given, so a Voronoi edge can still be built in time proportional to $\log n$ plus the number of its breakpoints.

Therefore, the first phase takes $O\big(K_d(\log m + \log^2 n) + |\mathrm{Vor}_P(S) \cap P(d)|\big)$ time, where K_d is the number of involved potential vertices. Note that for the Voronoi edges that intersect d, their breakpoints outside $P(d)$ will be counted in the respective triangles in $P'(d)$.

The second phase triangulates each cell in $\mathrm{Vor}_P(S) \cap P(d)$ and constructs the SPM from the associated site in each triangulated cell. Since Chazelle's algorithm [2] takes linear time to triangulate a simple polygon and Guibas et al's algorithm [5] takes linear time to build the SPM in a triangulated polygon, the second phase takes $O(|\mathcal{SD}' \cap P(d)|)$ time.

▶ Remark. Although all the sites lie outside $P(d)$, the information stored in $W_d\big(S'(d)\big)$ allows us not to conduct Guibas et al's algorithm from scratch. For example, for each anchor u, its anchor $a(u)$ is also stored, and the SPM edge of u is the line segment from u to the polygon boundary along the direction from $a(u)$ to u.

To sum up, a Propagate operation takes $O\big(K_d(\log m + \log^2 n) + |\mathcal{SD}' \cap P(d)|\big)$ time.

6 Subdivision construction

6.1 Construction of $\mathcal{SD}$

To construct $\mathcal{SD}$, we process each triangle $\triangle$ by the postorder traversal of the rooted partition tree $\mathcal{T}$ and build $\mathcal{SD} \cap \triangle$. We first assume that no diagonal of $\triangle$ is a polygon side, and we will discuss the other cases later. Let d be the root diagonal of $\triangle$ and adopt the convention in Section 2.2 and Section 2.3. When processing $\triangle$, since its two children, $\triangle(d_1)$ and $\triangle(d_2)$, have been processed, $W_{d_1}\big(S(d_1)\big)$ and $W_{d_2}\big(S(d_2)\big)$ are available.

The processing of each triangle $\triangle$ consists of 8 steps:

1. Initiate $\mathrm{Vor}_\triangle(S_\triangle)$ and $W_{(d,d_2)}(S_\triangle)$.
2. Extend $W_{d_1}\big(S(d_1)\big)$ into $\triangle$ to generate $W_{(d,d_2)}\big(S(d_1)\big)$ and construct $\mathrm{Vor}_P^*\big(S(d_1)\big) \cap \triangle$.
3. Merge $W_{(d,d_2)}(S_\triangle)$ and $W_{(d,d_2)}\big(S(d_1)\big)$ into $W_{(d,d_2)}\big(S(d_1) \cup S_\triangle\big)$ by which $\mathrm{Vor}_\triangle(S_\triangle)$ and $\mathrm{Vor}_P^*\big(S(d_1)\big) \cap \triangle$ are merged into $\mathrm{Vor}_P^*\big(S(d_1) \cup S_\triangle\big) \cap \triangle$.
4. Divide $W_{(d,d_2)}\big(S(d_1) \cup S_\triangle\big)$ into $W_d\big(S(d_1) \cup S_\triangle\big)$ and $W_{d_2}\big(S(d_1) \cup S_\triangle\big)$ along d_2.
5. Extend $W_{d_2}\big(S(d_2)\big)$ into $\triangle$ to generate $W_{(d_1,d)}\big(S(d_2)\big)$ and construct $\mathrm{Vor}_P^*\big(S(d_2)\big) \cap \triangle$.
6. Divide $W_{(d_1,d)}\big(S(d_2)\big)$ into $W_{d_1}\big(S(d_2)\big)$ and $W_d\big(S(d_2)\big)$ along d_1.
7. Insert $S_\triangle$ into $W_{d_1}\big(S(d_2)\big)$ to obtain $W_{d_1}\big(S(d_2) \cup S_\triangle\big)$.
8. Merge $W_d\big(S(d_1) \cup S_\triangle\big)$ and $W_d\big(S(d_2)\big)$ into $W_d\big(S(d)\big)$ by which $\mathrm{Vor}_P^*\big(S(d_1) \cup S_\triangle\big) \cap \triangle$ and $\mathrm{Vor}_P^*\big(S(d_2)\big) \cap \triangle$ are merged into $\mathrm{Vor}_P^*\big(S(d)\big) \cap \triangle = \mathcal{SD} \cap \triangle$.

We remark that $W_{d_1}\big(S(d_2) \cup S_\triangle\big)$ and $W_{d_2}\big(S(d_1) \cup S_\triangle\big)$ will be used to construct $\mathcal{SD}'$.

If exactly one diagonal d of $\triangle$ is not a polygon side, it is either the root triangle ▲ or a leaf triangle. For the former, compute $\mathrm{Vor}_\blacktriangle(S_\blacktriangle)$, extend $W_d(S(d))$ into ▲ to build $\mathrm{Vor}_P^*(S(d)) \cap \blacktriangle$, and merge $\mathrm{Vor}_\blacktriangle(S_\blacktriangle)$ and $\mathrm{Vor}_P^*(S(d)) \cap \blacktriangle$ into $\mathrm{Vor}_P^*(S) \cap \blacktriangle = \mathcal{SD} \cap \blacktriangle$; for the latter, compute $\mathrm{Vor}_\triangle(S_\triangle)$ and initiate $W_d(S_\triangle)$. If exactly two diagonals, d and d', of $\triangle$ are not polygon sides, where d is the root diagonal, then compute $\mathrm{Vor}_\triangle(S_\triangle)$ to initiate $W_d(S_\triangle)$ and $W_{d'}(S_\triangle)$, extend $W_{d'}\big(S(d')\big)$ into $\triangle$ to obtain $W_d\big(S(d')\big)$ and $\mathrm{Vor}_P^*\big(S(d')\big) \cap \triangle$, and merge $W_d(S_\triangle)$ and $W_d\big(S(d')\big)$ to obtain $W_d\big(S(d)\big)$ and $\mathrm{Vor}_P^*\big(S(d)\big) \cap \triangle = \mathcal{SD} \cap \triangle$.

By the operation times in Section 5, the time to process $\triangle$ is summarized as follows.

▶ **Lemma 3.** *It takes* $O\big((|S_\triangle| + K_\triangle)(\log m + \log^2 n) + I_\triangle\big)$ *plus amortized* $O(A_\triangle + 1)$ *time to process* $\triangle$, *where* $K_\triangle$ *is the number of involved potential vertices,* $I_\triangle$ *is the number of created diagram vertices, and* $A_\triangle$ *is the number of visited anchors in Steps 4 and 6.*

To apply Lemma 3, we bound $\sum_\triangle K_\triangle$, $\sum_\triangle I_\triangle$ and $\sum_\triangle A_\triangle$ by the following two lemmas.

▶ **Lemma 4.** *In the construction of* $\mathcal{SD}$, $\sum_\triangle K_\triangle = O(m)$ *and* $\sum_\triangle I_\triangle = O(n + m)$.

Proof. There are 6 intermediate diagrams: $\mathrm{Vor}_\triangle(S_\triangle)$ (step 1), $\mathrm{Vor}_P^*\big(S(d_1)\big) \cap \triangle$ (step 2), $\mathrm{Vor}_P^*\big(S(d_1) \cup S_\triangle\big) \cap \triangle$ (step 3), $\mathrm{Vor}_P^*\big(S(d_2)\big) \cap \triangle$ (step 5), $\mathrm{Vor}_P^*\big(S(d_2) \cup S_\triangle\big) \cap \triangle$ (step 7), and $\mathrm{Vor}_P^*\big(S(d)\big) \cap \triangle = \mathcal{SD} \cap \triangle$ (step 8). First, a potential vertex arises due to the formation of an incomplete Voronoi edge, and an incomplete Voronoi edge generates $O(1)$ potential vertices. For the first diagram, the total number of Voronoi edges is $O(\sum_\triangle |S_\triangle|) = O(|S|) = O(m)$. For each of the other 5 diagrams, we can define borders in a similar way to $\mathcal{SD}$ and thus obtain a subdivision of P. Since each site has at most one cell in each resultant subdivision, Euler's formula implies that each subdivision contains $O(m)$ Voronoi edges among cells, leading to the conclusion that $\sum_\triangle K_\triangle = O(m)$. Second, a created diagram vertex must be a vertex of the first diagram or the other 5 subdivisions. Since there are m sites, the first diagram results in $O(m)$ vertices for all the triangles, and by the same reasoning of [10, Lemma 3], each subdivision has $O(n + m)$ vertice, leading to that $\sum_\triangle I_\triangle = O(n + m)$. ◀

▶ **Lemma 5.** *In the construction of* $\mathcal{SD}$, $\sum_\triangle A_\triangle = O(n + m)$.

Proof. We consider Step 4 (divide $W_{(d,d_2)}\big(S(d_1) \cup S_\triangle\big)$ along d_2), which is similar to Step 6. Since there are $O(n + m)$ anchors, it is sufficient to bound the number of anchors that are visited by Step 4 but still involved in the future construction of $\mathcal{SD}$, namely $\mathcal{SD} \cap P'(d)$. Since the subcell of each visited anchor intersects d_2, if the subcell does not intersect d, its anchor will not be involved in constructing $\mathcal{SD} \cap P'(d)$. A subcell of a visited anchor intersects d in two cases. In the first case, the subcell contains v_2 and thus intersects both d and d_2. Since there are $O(n)$ triangles, the total number for the first case is $O(n)$. In the second case, the subcell intersects both d and d_2 but does not contain v_2. By the definition of $W_{(d,d_2)}\big(S(d_1) \cup S_\triangle\big)$, its associated "site" belongs to either $S_\triangle$ or $S(d_1)$. For the former, since its anchor must be the site itself, the total number is $\sum_\triangle S_\triangle = m$. For the latter, since all the sites in $S(d_1)$ lie outside $\triangle$, only one site in $S(d_1)$ can own a cell intersecting both d and d_2. Moreover, due to the visibility of a subcell, only one subcell in such a cell can intersect both d and d_2. Since there are $O(n)$ triangles, the total number is $O(n)$. ◀

By Lemma 3, 4, and 5, we conclude the construction time of $\mathcal{SD}$ as follows.

▶ **Theorem 6.** $\mathcal{SD}$ *can be constructed in* $O\big(n + m(\log m + \log^2 n)\big)$ *time.*

Proof. By Lemma 3, we need to bound $\sum_\triangle \big((|S_\triangle| + K_\triangle) \cdot (\log m + \log^2 n) + I_\triangle + A_\triangle + 1\big)$. It is trivial that $\sum_\triangle |S_\triangle| = |S| = m$ and $\sum_\triangle 1 = O(n)$. By Lemma 4 and Lemma 5, $\sum_\triangle K_\triangle = O(m)$, $\sum_\triangle I_\triangle = O(n+m)$, and $\sum_\triangle A_\triangle = O(n+m)$, leading to the statement. ◀

6.2 Construction of $\mathcal{SD}'$

To construct $\mathcal{SD}'$, we processes each triangle by the preorder traversal of the partition tree $\mathcal{T}$. For the root triangle $\blacktriangle$, let $\tilde{d}$ be its diagonal that is not a polygon side, build $W_{\tilde{d}}(S_\blacktriangle) = W_{\tilde{d}}\big(S'(\tilde{d})\big)$, and if $S_\blacktriangle = S$, further propagate $W_{\tilde{d}}(S_\blacktriangle)$ into $P(\tilde{d})$. For other triangles $\triangle$, we assume that neither $\triangle$ nor its parent $\triangle'$ has a polygon side; the other cases can be processed in a similar way. Since $\triangle'$ has been processed, $W_d\big(S'(d)\big)$ is available, and by the construction of $\mathcal{SD}$, $W_{d_2}\big(S(d_1) \cup S_\triangle\big)$ and $W_{d_1}\big(S(d_2) \cup S_\triangle\big)$ have been generated.

If $S(d) \neq \emptyset$, $\triangle$ is processed by the following 4 steps:

1. Extend $W_d\big(S'(d)\big)$ into $\triangle$ to obtain $W_{(d_1, d_2)}(S'(d))$ and $\mathrm{Vor}_P^*\big(S'(d)\big) \cap \triangle = \mathcal{SD}' \cap \triangle$.
2. If neither $S(d_1)$ nor $S(d_2)$ is empty, split $W_{(d_1, d_2)}(S'(d))$ into $W_{d_1}(S'(d))$ and $W_{d_2}(S'(d))$; otherwise, if $S(d_1)$ (resp. $S(d_2)$) is empty, divide $W_{(d_1, d_2)}(S'(d))$ into $W_{d_1}(S'(d))$ and $W_{d_2}(S'(d))$ along d_1 (resp. d_2).
3. Join $W_{d_1}\big(S(d_2) \cup S_\triangle\big)$ and $W_{d_1}(S'(d))$ into $W_{d_1}(S'(d_1))$, and join $W_{d_2}\big(S(d_1) \cup S_\triangle\big)$ and $W_{d_2}(S'(d))$ into $W_{d_2}(S'(d_2))$.
4. If $S(d_1)$ (resp. $S(d_2)$) is empty, propagate $W_{d_1}(S'(d_1))$ (resp. $W_{d_2}(S'(d_2))$) into $P(d_1)$ (resp. $P(d_2)$) to build $\mathcal{SD}' \cap P(d_1)$ (resp. $\mathcal{SD}' \cap P(d_2)$).

We first analyze Split and Propagate operations.

▶ **Lemma 7.** *The total number of Split operations is $O(m)$.*

Proof. A Split operation occurs only if neither $S(d_1)$ nor $S(d_2)$ is empty. We recursively remove leaf nodes (triangles) containing no site from $\mathcal{T}$, so that each leaf node in the resulting tree $\mathcal{T}'$ contains a site. Then a Split operation occurs in an internal node of $\mathcal{T}'$ with two children. Since there are m sites, $\mathcal{T}'$ has $O(m)$ leaf nodes, and since $\mathcal{T}'$ is a binary tree, the number of internal nodes with two children is $O(m)$, leading to $O(m)$ Split operations. ◀

▶ **Lemma 8.** *The total time for all the Propagate operations is $O\big(n + m(\log m + \log^2 n)\big)$.*

Proof. For a sub-polygon to propagate, since each edge of the resultant subdivision has a vertex in the sub-polygon, the number of created edges is bounded by the number of created vertices. Therefore, by Section 5.4, the total time is $O\big(K(\log m + \log^2 n) + I\big)$, where K is the number of involved potential vertices and I is the number of created diagram vertices. Moreover, by the same reasoning of Lemma 4, $K = O(m)$, and since all the sub-polygons to propagate are disjoint, $I = O(|\mathcal{SD}'|) = O(m + n)$, leading to the statement. ◀

By Lemma 7, Lemma 8 and the reasoning of Theorem 6, we conclude the construction time of $\mathcal{SD}'$ as follows.

▶ **Theorem 9.** *$\mathcal{SD}'$ can be constructed in $O\big(n + m(\log m + \log^2 n)\big)$ time.*

References

1   Boris Aronov. On the geodesic Voronoi diagram of point sites in a simple polygon. *Algorithmica*, 4(1-4):109–140, 1989.
2   Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991.
3   Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.
4   Leonidas J. Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, 1989.
5   Leonidas J. Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987.
6   John Hershberger. A new data structure for shortest path queries in a simple polygon. *Information Processing Letters*, 38(5):231–235, 1991.
7   Kurt Mehlhorn and Peter Sanders. Sorted sequences. In *Algorithms and Data Structures: The Basic Toolbox*. Springer-Verlag Berlin Heidelberg, 2008.
8   Joseph S. B. Mitchell. Geometric shortest paths and network optimization. In *Handbook of Computational Geometry*, pages 633–701. Elsevier, 2000.

9   Eunjin Oh and Hee-Kap Ahn. Voronoi diagrams for a moderate-sized point-set in a simple polygon. In *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, pages 52:1–52:15, 2017.

10   Evanthia Papadopoulou and D. T. Lee. A new approach for the geodesic Voronoi diagram of points in a simple polygon and other restricted polygonal domains. *Algorithmica*, 20(4):319–352, 1998.

11   Robert Endre Tarjan. Updating a balanced search tree in O(1) rotations. *Information Processing Letters*, 16(5):253–257, 1983.

12   Robert Endre Tarjan. Efficient Top-Down Updating of Red-Black Trees. Technical report, Technical Report TR-006-85. Dapartment of Computer Science, Princeton University, 1985.