Chapter 12

Distance Oracles

In this Chapter we discuss the data structure version of the shortest paths problem in planar graphs. We will show ways to preprocess a planar graph to produce representations that support efficient vertex-to-vertex distance queries. Such a representation is called a *distance oracle*. A good distance oracles requires small space, answers queries quickly, and can be constructed quickly. We shall show different data structures achieving different tradeoffs between space and query time. There is no known lower bound against an oracle with $O(n)$ space and construction time that can answer distance queries exactly in $O(1)$ time. In Section 12.1 we describe, for any $\epsilon > 0$, an oracle for undirected plane graphs with $O(n\epsilon^{-1} \log n)$ space and $O(n\epsilon^{-1} \log^2 n)$ construction time, that returns a $(1+\epsilon)$-multiplicative approximation for the distance between any two vertices in the graph. In Section 12.2 we describe an exact distance oracle for directed plane graphs with $\tilde{O}(n)$ space and construction time, and $\tilde{O}(\sqrt{n})$ query-time. Here, $\tilde{O}(\cdot)$ hides polylogarithmic factors in n. In Section 12.3 we describe an exact oracle for directed plane graphs with $\tilde{O}(n^{4/3})$ space and $O(\log^2 n)$ query time.

We start with a planar embedded graph G with nonnegative edge-lengths. Since we are dealing with distances, we can assume no self-loops and no parallel edges. We can also assume that G is triangulated, by adding infinite-length edges as needed to ensure that each face is a triangle.

All efficient distance oracles for planar graphs rely in some way or another on a decomposition of the graph using separators. Recall the concept of a decomposition tree of a plane graph G (Definition 5.9.5). We consider a specific decomposition tree $\mathcal{T}$ of G which we will refer to as a *recursive decomposition* of G. The root of $\mathcal{T}$ corresponds to G, and the two children x_0, x_1 of each internal node $x \in \mathcal{T}$ are obtained by separating the region R_x using a simple cycle separator S_x. (The precise choices of S_x will be specified later.) The region R_{x_0} corresponding to x_0 is the subgraph of R_x enclosed by S_x, and R_{x_1} is the subgraph of R_x not strictly enclosed by S_x. Note that S_x is a face of both R_{x_0} and R_{x_1}. The decomposition is *complete* if the leaf subgraphs are small enough according to some measure $m(\cdot)$ of graph size, i.e. for every leaf x, $m(R_x) \leq c$

for a constant c.

We say a path P *crosses* a simple cycle C if P contains both an edge strictly enclosed by C and an edge not enclosed by C.

Lemma 12.0.1. *Consider a recursive decomposition $\mathcal{T}$ of G using simple cycle separators. Let P be a path in G. Let x, y be nodes of $\mathcal{T}$. If P crosses S_x and S_y then then P crosses S_w for some common ancestor w of x and y.*

It follows that

Corollary 12.0.2. *For any path P, there is a unique rootmost node z in $\mathcal{T}$ such that P crosses $S(z)$.*

This property is crucial in some of the oracles we describe.

12.1 An approximate distance oracle for undirected planar graphs

In this section we describe an approximate distance oracle for undirected planar graphs. For approximate distances, we have another input, a parameter $\epsilon > 0$. A query DISTANCE(u, v) will be answered with the length of a u-to-v path in G such that the length is at most $1 + \epsilon$ times the u-to-v distance.

12.1.1 Overall strategy

One key tool is the fundamental-cycle separator of edges (Lemma 5.3.3):

Lemma: *There is a linear-time algorithm that, given a triangulated plane graph G with a $\frac{1}{3}$-proper assignment of weights to edges, and a spanning tree T, returns a nontree edge $\hat{e}$ such that the fundamental cycle of $\hat{e}$ with respect to T is a $\frac{2}{3}$-balanced cycle separator for G.*

Let G be the input graph. The algorithm computes a shortest path tree T rooted at an arbitrary vertex r of G. It then uses fundamental cycle separators with respect to T to obtain a complete recursive decomposition $\mathcal{T}$ of the input graph G. Each separator in the decomposition is a fundamental cycle C. Note that C consists of (1) a nontree edge uv, together with (2) the $\text{lca}_T(u, v)$-to-u path in T, and (3) the $\text{lca}_T(u, v)$-to-v path in T. This implies that there are two leafward paths such that every vertex on the separator lies on at least one of the paths (the least common ancestor lies on both). Note that since T is a shortest path tree, both leafward paths are shortest paths in G.

The strategy for shortest-path approximation is as follows. Let u and v be two given vertices. Assume u and v do not belong to the same leaf region of $\mathcal{T}$ (distances between vertices in the same leaf region of $\mathcal{T}$ can be tabulated using linear space). Let w be the leafmost node of $\mathcal{T}$ such that $u, v \in R_w$. Let P be a shortest u-to-v path in G. Let z be the rootmost node of $\mathcal{T}$ such that P crosses S_z. Note that z exists by Lemma 12.0.2, and that z is an ancestor of w. Also

note that the algorithm does not immediately know z, given u and v. For each ancestor y of w, the algorithm will estimate the minimum length of a path that (i) crosses S_y but (ii) does not cross $S_{y'}$ for any proper ancestor of y in $\mathcal{T}$. The estimate produced when $y = z$ will satisfy the requirements.

12.1.2 Connections to a shortest path

Therefore we turn to the problem of estimating paths that cross a separator S_y. We will use the fact that the vertices of S_y belong to two shortest paths.

Let H be a planar embedded graph, let S be a fundamental-cycle separator, and let $\mathcal{P}$ be the set consisting of the two paths comprising S. For each path $P \in \mathcal{P}$, for each vertex v of $H - S$, we will select a set of vertices of $V(P)$, which we call the *connectees* of v (and P). For each connectee r of v we will record the r-to-v distance in H. We will call the pair (r, v) a *connection* for v. Our construction will have the following two properties.

1. For each vertex v, the number of connections for v is at most $8/(\epsilon - \epsilon^2)$.

2. For any two vertices u, v, a shortest u-to-v path in H that crosses S is ϵ-approximated by the shortest u-to-v path in H that goes from u to a connectee r_u for u, then along a path $P \in \mathcal{P}$ containing r_u to a connectee for v, then to v.

Now we give an algorithm that, for a given vertex v and a path P, selects some vertices of P to be connectees of v. Let r_0 be the vertex of P that is closest to v among all vertices of P. For every node r of P, define $h[r] = dist(r, r_0)$. The algorithm designates r_0 as a connectee for v. It then uses two phases, a forward phase and a backward phase, to select more connectees. The forward phase selects connectees $r_1, r_2, \ldots$ and the backward phase selects connectees $r_{-1}, r_{-2}, \ldots$.

We describe the forward phase. The backward phase in analogous. The forward phase considers vertices r of P one by one, in leafward order.

for $i = 0, 1, 2, \ldots$
  let r be the first vertex of P after r_i such that
    $(1 + \epsilon)dist(v, r) < dist(v, r_i) + h[r] - h[r_i]$
$P = dist(r_i, r)$.
  $r_{i+1} := r$
until there is no such vertex r

Since r appears after r_i on P, $h[r] - h[r_i] = dist(r_i, r)$. Hence, the expression $dist(v, r_i) + h[r] - h[r_i]$ is the length of an indirect path from v to r which goes via a shortest path to r_i, thence along P to r. Thus the condition in the procedure holds if the direct shortest v-to-r path is much shorter than the indirect path.

Let $r_1, \ldots, r_k$ be the connectees chosen by the forward phase. We say that a vertex v is *covered* by r_i if $dist(v, r_i) + dist(r_i, r) \leq (1 + \epsilon)dist(v, r)$.

Lemma 12.1.1. *For any vertex r of P that is leafward of r_0, there is a connectee r_i that is rootward of r such that*

$$dist(v, r_i) + dist(r_i, r) \leq (1 + \epsilon) dist(v, r) \tag{12.1}$$

Proof. Let r_i be the last vertex of P designated a connectee before a vertex after r was considered. If $r_i = r$ then (12.1) holds trivially. If not, the inequality in the procedure did not hold at the time r was considered, so we have

$$(1 + \epsilon) dist(v, r) \geq dist(v, r_i) + h[r] - h[r_i]$$

which is equivalent to (12.1). □

Lemma 12.1.2. *The number k of connectees chosen by the forward phase is less than $2/(\epsilon - \epsilon^2)$.*

Proof. By Taylor series expansion, $(1 + \epsilon)^{-1} < 1 - (\epsilon - \epsilon^2)$. The choice of r_{i+1} guarantees

$$\begin{aligned} dist(v, r_{i+1}) &< (1 + \epsilon)^{-1} \left(dist(v, r_i) + h[r_{i+1}] - h[r_i] \right) \\ &\leq (1 + \epsilon)^{-1} dist(v, r_i) + h[r_{i+1}] - h[r_i] \\ &\leq dist(v, r_i) - (\epsilon - \epsilon^2) dist(v, r_i) + h[r_{i+1}] - h[r_i] \\ &\leq dist(v, r_i) - (\epsilon - \epsilon^2) dist(v, r_0) + h[r_{i+1}] - h[r_i] \end{aligned} \tag{12.2}$$

Therefore we obtain the recurrence relation

$$dist(v, r_{i+1}) - h[r_{i+1}] < dist(v, r_i) - h[r_i] - (\epsilon - \epsilon^2) dist(v, r_0)$$

which yields

$$dist(v, r_i) - h[r_i] < dist(v, r_0) - h[r_0] - i(\epsilon - \epsilon^2) dist(v, r_0)$$

Recall that $h[r_i] = dist(r_0, r_i)$. Using the fact that $h[r_i] = 0$, we have

$$dist(v, r_i) - h[r_i] < dist(v, r_0) - i(\epsilon - \epsilon^2) dist(v, r_0) \tag{12.3}$$

By the triangle inequality, the r_0-to-r_i distance is at most the r_0-to-v distance plus the v-to-r_i distance, so

$$h[r_i] \leq dist(v, r_0) + dist(v, r_i)$$

which is equivalent to

$$dist(v, r_i) - h[r_i] \geq -dist(v, r_0)$$

which, combined with (12.3), yields

$$-dist(v, r_0) < dist(v, r_0) - k(\epsilon - \epsilon^2) dist(v, r_0)$$

where k is the number of connectees chosen by the forward phase. We therefore obtain

$$k < 2/(\epsilon - \epsilon^2)$$

□

Remark: In the proof of Lemma 12.1.2, we used (12.2), which is weaker than the inequality actually used in the procedure. We could replace the inequality used in the procedure with (12.2), and Lemma 12.1.2 would still hold. Lemma 12.1.1 would also still hold.

12.1.3 The oracle

Data structure. The data structure consists of the following:

1. A shortest path tree T of G, rooted arbitrarily, the distances $h[\cdot]$ from the root, and a complete recursive decomposition $\mathcal{T}$ using fundamental cycle separators with respect to T.
2. For each vertex v, a leaf of $\mathcal{T}$ containing v.
3. For every leaf node $x \in \mathcal{T}$, the pairwise distances between all vertices of R_x.
4. For every non-leaf node $x \in \mathcal{T}$, for each of the two paths P comprising S_x, for each internal vertex $v \in R_x$, a list of the connectees of v and P in leafward order, as well as the distance $dist_{R_x}(v, r)$ betwwen each connectee r and v. Here, a vertex $v \in R_x$ is called internal to R_x if it does not belong to $S_{x'}$ for any strict ancestor x' of x in $\mathcal{T}$.

Storing items 1-3 requires $O(n)$ space. For every level ℓ of $\mathcal{T}$, each vertex v of G is internal to at most a single region R_x at level ℓ. Hence, by Lemma 12.1.2 and since the depth of $\mathcal{T}$ is $O(\log n)$, storing item 4 requires $O(\epsilon^{-1} n \log n)$ space.

Query. Given u, v, the query algorithm retrieves nodes $x, y \in \mathcal{T}$ such that $u \in R_x$ and $v \in R_y$. If $R_x = R_y$, the algorithm returns $dist(u, v)$, which is stored explicitly in item 1. Otherwise, the algorithm finds $w = \text{lca}_{\mathcal{T}}(x, y)$. For every ancestor y of w in $\mathcal{T}$, for each of the two paths P comprising S_w, the algorithm computes a distance estimate as follows. For each connectee r of u on P, let $t^-(r)$ and $t^+(r)$ be the conectees of v that precede r and follow r in the linear ordering of the conectees of u and of v on P along P, respectively. Define for each conectee t of v on P, the connectees $r^-(t)$ and $r^+(t)$ analogously. These conectees can be identified at query in $O(\epsilon^{-1})$ time by traversing the list of connectees of u and v in an algorithm similar to that of merging two sorted lists. The algorithm then computes

$$\min_r \begin{cases} dist_{R_y}(u, r) + dist_{(}r, t^-(r)), dist_{R_y}(t^-(r), v) \\ dist_{R_y}(u, r) + dist_{(}r, t^+(r)), dist_{R_y}(t^+(r), v) \end{cases},$$

and

$$\min_t \begin{cases} dist_{R_y}(u, r^-(t)) + dist_{(}r^-(t), t), dist_{R_y}(t, v) \\ dist_{R_y}(u, r^+(t)) + dist_{(}r^+(t), t), dist_{R_y}(t, v) \end{cases},$$

where the minimization is over all connectees r of u and t of v on P, respectively. The algorithm returns the minimum distance estimate found.

Since each distance estimate corresponds to some path in G, the reported distance is at least $dist_G(u, v)$. Let Q be a shortest u-to-v path. Let z be the rootmost node of $\mathcal{T}$ such that Q crosses S_z. Note that z exists by Lemma 12.0.2, and that z is an ancestor of w. Let p be a vertex on $Q \cap P$. By Lemma 12.0.2 and by Lemma 12.1.1, for $y = z$, for r a connectee of u that precedes or follows p on P, and for t a connectee of u that precedes or follows p on P,

$$\begin{aligned} dist_{R_z}(u, r) + dist(r, t) + dist_{R_z}(v, t) &\leq \\ dist_{R_z}(u, r) + dist(r, p) + dist(t, p) + dist_{R_z}(v, t) &\leq \\ (1 + \epsilon)(dist_{R_z}(u, p) + dist_{R_z}(v, p)) &= dist(u, v), \end{aligned}$$

which proves the correctness of the query.

By Lemma 12.1.2, the time to compute the distance estimate for a particular level ℓ is $O(\epsilon^{-1})$. Since the depth of $\mathcal{T}$ is $O(\log n)$, the total query time is $O(\epsilon^{-1} \log n)$.

12.1.4 Efficient construction

We now show how to construct the oracle in $O(n\epsilon^{-1} \log^2 n)$ time. It is easy to contsruct parts 1–3 of the data structure in $O(n \log n)$ time. The challenging part is in computing the connections for part 4 of the data structure. We give an algorithm that, for a subgraph H of G, and a shortest path P in H, finds a set of connections satisfying the properties stated in Section 12.1.2 for all vertices of H such that each vertex v has $4/(\epsilon - \epsilon^2)$ connections. The running time of this algorithm is $O(n\epsilon^{-1} \log n)$, so invoking it on each region in the complete decomposition tree computes all the information required for part 4 of the data structure in $O(n\epsilon^{-1} \log^2 n)$ time.

The algorithm trims the graph along the path P (see Section 4.9). In the resulting graph, which we also denote by H, the former darts of P form a new face. Note that, because P is a shortest path, this transformation does not change any the shortest paths and distances from any vertex v to any vertex of P.

The algorithm applies the MSSP algorithm of Chapter 7 to (the modified) H with the new face corresponding to P as the distinguished face, and finds, for each $0 \leq i < k$, the sequence A_i of pivots that transform the r_i-rooted shortest-path tree into the r_{i+1}-rooted shortest-path tree (ordered so that each intermediate result is still a tree). Each pivot is represented by a triple (uw, α, vw) where uw is the arc to be removed, vw is the arc to be inserted, and α is the decrease in the distance to the w-rooted subtree.

The algorithm constructs an auxiliary graph from H by adding an artificial root $\hat{r}$ and zero-length arcs $\hat{r}r_i$ to the vertices r_i of P. Next, the algorithm finds a shortest-path tree $\hat{T}$ of the auxiliary graph rooted at $\hat{r}$. For each i, let $\hat{T}_i$ be the subtree of $\hat{T}$ rooted at r_i. For each vertex $v \neq \hat{r}$, let $\hat{\imath}(v)$ denote that integer i such that v belongs to $\hat{T}_i$. That is, $dist(r_{\hat{\imath}(v)}, v) = \min_i dist(r_i, v)$.

The algorithm next performs two phases, a forward phase and a backward phase. In each phase, the algorithm designates pairs $(r, v) \in V(P) \times V(H)$ as

connections. The output of the algorithm is the set of all pairs designated as connections. We describe the forward phase; the backward phase is similar.

At any point in the running of the phase, for a vertex v, let $r(v)$ denote the vertex r such that (r, v) was the last connection designated for v, or $r(v) = \perp$ if no connection has yet been designated during the phase.

The algorithm maintains a link-cut tree representation of a tree T of H. The link-cut tree supports costs assigned to vertices, with descendant bulk updates and descendant searches. The cost of v is denoted $\sigma(v)$.

The link-cut tree also maintains for each vertex v a label $\boldsymbol{d}[v]$ satisfying

$$\boldsymbol{d}[v] = \text{root-to-}v \text{ distance in } T \tag{12.4}$$

The algorithm maintains the following invariant. Let r be the root of T. For each vertex v, if $r(v) \neq \perp$ then

$$\sigma(v) = (1+\epsilon)\boldsymbol{d}[v] - (dist(r, r(v)) + dist(r(v), v)) \tag{12.5}$$

Therefore, if $\sigma(v)$ is nonpositive, the need to cover v suggests that (r, v) be designated a connection.

```
initialize T to be the r_0-rooted shortest-path tree
for each vertex v, initialize d[v] to be the length of the root-to-v path
initialize σ(v) := ∞ for every vertex v
for i := 0, 1, 2, ..., k,
1   for each vertex v in T̂_i,
2       designate (r_i, v) a connection
3       assign σ(v) := ε d[v]
4   while there exists a vertex v with σ(v) < 0,
5       designate (r_i, v) a connection
6       assign σ(v) := ε d[v]
7   if i < k,
8       comment: reroot the tree by r_{i+1}
9       remove the arc of T entering r_{i+1} and add the arc r_{i+1} r_i
11      for every v, increase d[v] by ℓ(r_{i+1} r_i)
12      set d[r_{i+1}] := 0
13      for every v, increase σ(v) by εℓ(r_{i+1} r_i)
14      comment: Carry out pivots.
15      for each (uw, α, vw) ∈ A_i,
16      remove uw from T and insert vw
17      subtract α from d[w'] for every vertex w' in the w-rooted tree
18      subtract (1+ε)α from σ(w') for every vertex w' in the w-rooted tree
```

Correctness

The algorithm ensures that, for every vertex v, equations (12.4) and (12.5) hold. These hold immediately after the initializations. In Line 2 and in Line 5, a new connection is designated for v. The assignment to $\sigma(v)$ in Lines 3 and 6

preserve the invariant (12.5). After the change of root in Line 9, Line 11 and 13 restore (12.4) and (12.5) for every vertex v except r_{i+1}. Line 12 restores (12.4) for $v = r_{i+1}$. Since r_{i+1} belongs to $\hat{T}_{i+1}$, $r(r_{i+1}) = \perp$ so the invariant does not require 12.5 to hold for $v = r_{i+1}$.

The pivots in Lines 15-16 change the tree T, but the updates in Lines 17 and 18 restore (12.4) and (12.5).

Claim: The forward phase ensures that, after iteration i, for each vertex v, if $i \geq \hat{i}(v)$ then there is a connection (r_j, v) such that $(1+\epsilon)dist(r_i, v) \geq dist(r_i, r_j) + dist(r_j, v)$.

Proof. If at the beginning of iteration i we have

$$(1+\epsilon)dist(r_i, v) < dist(r_i, r(v)) + dist(r(v), v)$$

then v is selected in some iteration of the while-loop of Line 4, and (r_i, v) is designated a connection. □

Running time

Each iteration of the while-loop in Line 4 takes amortized $O(\log n)$ time. The number of iterations is the total number of connections established, which is $O(n\epsilon^{-1})$. Lines 9-13 takes $O(\log n)$, and these are executed $k \leq n$ times, for a total of $O(n \log n)$ time. Each execution of Lines 16-18 takes $O(\log n)$ time. Over the course of the whole phase, the number of iterations of the loop in Line 15 is $O(m)$, which is $O(n)$, so the total time for Lines 15-18 over the course of the algorithm is $O(n \log n)$. Thus the total time is $O(n\epsilon^{-1} \log n)$.

12.2 An Exact distance oracles with $\tilde{O}(n)$ space and $\tilde{O}(\sqrt{n})$ query time

In this section we turn our attention to exact distance oracles. We start with an oracle with nearly linear space that can answer distance queries exactly in $\tilde{O}(\sqrt{n})$ time. Let G be a directed plane graph G with non-negative arc lengths. The oracle consists of a complete recursive decomposition $\mathcal{T}$ of G using small simple cycle separators (Theorem 5.8.1). Each vertex $v \in G$ stores a pointer to a leaf node $x \in \mathcal{T}$ such that R_x contains v. For each node x of $\mathcal{T}$ other than the root r, let y denote the parent of x. Recall that the separator cycle S_y is a face of the region R_x. The oracle stores the MSSP data structure (Section 7.9.3) for the face S_y in the region R_x. In addition, it stores DDG_x, the dense distance graph of S_x in R_x, Recall from Chapter 8 that this is the compete graph on the vertices of S_x, where the length of each arc ww' is the w-to-w' distance in R_x. This complete graph is represented by a weighted adjacency matrix. The MSSP data structure requires $O(|R_x| \log |R_x|)$ space and preprocessing time. Since $|S_x| = O(\sqrt{|R_x|})$, storing DDG_x requires $O(|R_x|)$ space, and it can be computed in $O(|R_x| \log |R_x|)$ time from the MSSP data structure. Since the

total size of all regions corresponding to nodes at the each level of $\mathcal{T}$ is $O(n)$, and since $\mathcal{T}$ has $O(\log n)$ levels, the total size and construction time of the data structure are $O(n \log^2 n)$.

We now describe how to answer a query for the distance from u to v. Let x and y be leaf regions of $\mathcal{T}$ such that $u \in R_x$ and $v \in R_y$, respectively. If $x = y$ the algorithm computes in constant time the distance $dist_{R_x}(u, v)$ within the constant size region R_x. Let z' be the lowest common ancestor of x and y in $\mathcal{T}$. For each ancestor z of z' that is not a leaf of $\mathcal{T}$ (z' is a leaf only when $x = y$), the query algorithm computes $dist_{R_z}(u, v)$ using the following lemma.

Lemma 12.2.1. *$dist_{R_z}(u, v)$ can be computed in $O(|S_z| \log^2 |S_z|)$ time.*

Proof. Let z_0 and z_1 be the children of z in $\mathcal{T}$ such that $u \in R_{z_0}$ and $v \in R_{z_1}$. Observe that any u-to-v path that is restricted to R_z intersects S_z. Such a path can be decomposed into (i) a path in R_{z_0} from u to a vertex of S_z, (ii) zero or more paths whose endpoints belong to S_z, each of which is either in R_{z_0} or in R_{z_1}, and (iii) a path in R_{z_1} from a vertex of S_z to v. The algorithm runs FR-Dijkstra (Chapter 9) on the dense distance graph of z with respect to S_z, which consists of the two cliques DDG_{z_0} and DDG_{z_1}. It initializes the distance labels for FR-Dijkstra with the distances in R_{z_0} from u to the vertices of $S_{z'}$ (which can be queried in $O(\log |R_z|) = O(\log |S_z|)$ time per distance from the MSSP data structure stored for z_0). Since the number of vertices of each DDG_{z_i} is $|S_z|$, the running time of FR-Dijkstra is $O(|S_z| \log^2 |S_z|)$. Thus, FR-Dijkstra outputs the distance in R_z from u to the vertices of S_z. The algorithm then computes $dist_{R_z}(u, v)$ as

$$\min_{w \in S_z} dist_{R_z}(u, w) + distR_{z_1}(w, v).$$

Note that the distance $distR_{z_1}(w, v)$ for any $w \in S_z$ can be queried in $O(\log |R_z|) = O(\log |S_z|)$ time from the MSSP data structure stored for z_1. □

The algorithm returns the minimum $dist_{R_z}(u, v)$ found among all ancestors z of z'. The running time is dominated by the invocation of FR-Dijkstra at the top level when $z = r$, which takes $O(\sqrt{n} \log^2 n)$, because $|S_r| = O(\sqrt{n})$.

12.3 An exact oracle with $\tilde{O}(n^{4/3})$ space and $O(\log^2 n)$ query time

In this section we describe an exact distance oracle with space $\tilde{O}(n^{4/3})$, that can answer distance queries in $O(\log^2 n)$ time.

12.3.1 Additively weighted Voronoi diagrams.

Let H be a directed planar graph with real edge-lengths, and no negative-length cycles. Assume that all faces of H are triangles except, perhaps, a single face h, which we regard as the infinite face. Let S be the set of vertices that lie on

h. The vertices of S are called *sites*. Each site $s \in S$ has a weight $\omega(s) \geq 0$ associated with it. The additively weighted distance between a site $s \in S$ and a vertex $v \in V$, denoted by $d^{\omega}(s, v)$ is defined as $\omega(s)$ plus the length of the s-to-v shortest path in H.

Definition 12.3.1. *The additively weighted Voronoi diagram of (S, ω) (denoted $\mathsf{VD}(S, \omega, H)$) is a partition of $V(H)$ into pairwise disjoint sets, one set $\mathsf{Vor}(s)$ for each site $s \in S$. The set $\mathsf{Vor}(s)$, which is called the Voronoi cell of s, contains all vertices in $V(H)$ that are closer (w.r.t. $d^{\omega}(.,.)$ in H) to s than to any other site in S.*

There is a dual representation $\mathsf{VD}^*(S, \omega, H)$ (or simply VD^*) of a Voronoi diagram $\mathsf{VD}(S, \omega, H)$. Let H^* be the dual of H. Let VD_0^* be the subgraph of H^* induced the edges whose endpoints in H are in different Voronoi cells. Let VD_1^* be the graph obtained from VD_0^* by contracting edges incident to degree-2 vertices one after another until no degree-2 vertices remain. The vertices of VD_1^* are called Voronoi vertices. A Voronoi vertex $f^* \neq h^*$ is dual to a face f such that the vertices of H incident to f belong to three different Voronoi cells. We call such a face *trichromatic*. Each Voronoi vertex f^* stores for each vertex u incident to f the site s such that $u \in \mathsf{Vor}(s)$. Note that h^* (i.e. the dual vertex corresponding to the face h to which all the sites are incident) is a Voronoi vertex. Each face of VD_1^* corresponds to a cell $\mathsf{Vor}(s)$. Hence there are at most $|S|$ faces in VD_1^*. Since the minimum degree of a vertex in VD_1^* is 3 the sparsity lemma (Lemma 4.3.1) applies, so the complexity (i.e., the number of vertices, edges and faces) of VD_1^* is $O(|S|)$. Finally, we define VD^* to be the graph obtained from VD_1^* after replacing the node h^* by multiple copies, one for each occurrence of h as an endpoint of an edge in VD_1^*. See Figure 12.1 for an illustration.

Lemma 12.3.2. *If ω is such that every vertex of S lies in its own Voronoi Cell then $\mathsf{VD}^*(S, \omega, H)$ is a tree.*

Proof. Suppose that VD^* contains a cycle C^*. Since the degree of each copy of h^* is one, the cycle does not contain h^*. Therefore, since all the sites are on the boundary of the hole h, the vertices of H enclosed by C^* are in a Voronoi cell that contains no site, a contradiction.

To prove that VD^* is connected, observe that in VD_1^*, every Voronoi cell is a face (cycle) going through h^*. Let C^* denote this cycle. If C^* is disconnected in VD^* then, in VD_1^*, C^* has more than 2 edges incident to h^*. But this implies that the cell corresponding to C^* contains more than a single site, a contradiction. Thus, the boundary of every Voronoi cell is a connected subgraph of VD^*. For any i, consider the edge $e_i = s_i s_{i+1}$. Since the endpoints of e_i in H are in distinct Voronoi cells, $e_i \mathsf{VD}_0^*$. Therefore, the edge of VD_1^* into which e_i is contracted belongs to the two faces of VD_1^* that correspond to the $\mathsf{Vor}(s_i)$ and to $\mathsf{Vor}(s_{i+1})$. It follows that all the faces of VD_1^* are connected, and hence VD^* is connected. □

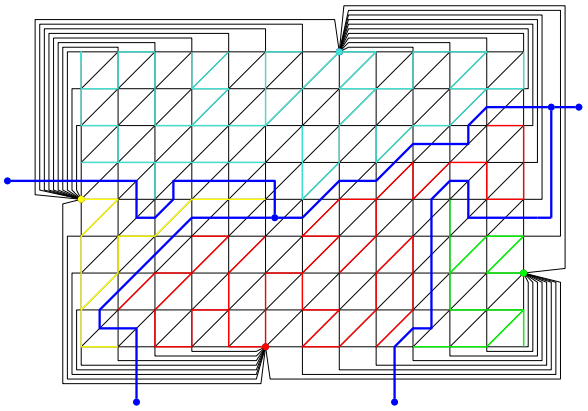

Figure 12.1: A planar graph (black edges) with four sites on the infinite face together with the dual Voronoi diagram VD^* (in blue). The sites are shown together with their corresponding shortest path trees (in turquoise, red, yellow, and green).

Our representation of $\mathsf{VD}(S, \omega, H)$ consists of the tree $\mathsf{VD}^*(S, \omega, H)$. In addition, each Voronoi vertex (i.e., each node of $\mathsf{VD}^*(S, \omega, H)$), corresponding to a face f of H stores, for each vertex v of f, the site $s \in S$ such that $v \in \mathsf{Vor}(s)$.

12.3.2 Point location in Voronoi diagrams

A *point location query* for a node v in a Voronoi diagram VD asks for the site s of VD such that $v \in \mathsf{Vor}(s)$ and for the additive distance from s to v. We describe a data structure supporting efficient point location, which is captured by the following theorem, which is proved in the remainder of this section.

Theorem 12.3.3. *Given an MSSP data structure for H with distinguished face h (see Section 7.9.3), and given $\mathsf{VD}^*(S, \omega, H)$, after $O(|S|)$-time preprocessing, point location queries can be answered in time $O(\log^2 |H|)$.*

Recall that H is triangulated (except the face h). For technical reasons that will be apparent later, we embed in every face f (other than h), with vertices y_1, y_2, y_3, three artificial auxiliary vertices y_j^f for $j = 1, 2, 3$, each with a single zero-length incident edge (y_j, y_j^f). The main idea is as follows. In order to find the Voronoi cell $\mathsf{Vor}(s)$ to which a query vertex v belongs, it suffices to identify an edge e^* of VD^* that is adjacent to $\mathsf{Vor}(s)$. Given e^* we can simply check which of its two adjacent cells contains v by comparing the distances from the corresponding two sites to v (distances from sites are available from the MSSP data structure). The point location structure is based on a *centroid decomposition* of the tree VD^* into connected subtrees, and on the ability to determine which of the subtrees is the one that contains the desired edge e^*.

The preprocessing consists of just computing a centroid decomposition of VD^*. A *centroid* of an n-node tree T is a node $u \in T$ such that removing u and replacing it with copies, one for each edge incident to u, results in a set of trees, each with at most $\frac{n+1}{2}$ edges. A centroid always exists in a tree with at least one edge. The centroid decomposition of VD^* is defined recursively. In every step of the centroid decomposition we work with a connected subtree T^* of VD^*. Initially, T^* is the entire tree VD^*. Recall that there are no nodes of degree 2 in VD^*. If there are no nodes of degree 3, then T^* consists of a single edge of VD^*, and the decomposition terminates. Otherwise, we choose a centroid c^*, and partition T^* into the three subtrees T_0^*, T_1^*, T_2^* obtained by splitting c^* into three copies, one for each edge incident to c^*. Since the size of VD^* is $O(|S|)$, the depth of this recursive decomposition is $O(\log |S|)$. Such a decomposition can be computed easily computed in $O(|S| \log |S|)$ time, and in fact can be computed in $O(|S|)$ time. It can be represented as a ternary tree which we call the *centroid decomposition tree*, in $O(|S|)$ space. Each non-leaf node of the centroid decomposition tree corresponds to a centroid vertex c^*, which is stored explicitly. We will refer to nodes of the centroid decomposition tree by their associated centroid. Each node also implicitly corresponds to the subtree of VD^* of which c^* is the centroid. The leaves of the centroid decomposition tree correspond to single edges of VD^*, which are stored explicitly.

Point location queries for a vertex v in the Voronoi diagram VD are answered by invoking procedure HandleCentroid with input (T^*, v), where T^* is the centroid decomposition tree of VD^*.

The procedure HandleCentroid gets as input a centroid decomposition tree T^* of a subtree of a Voronoi diagram VD^*, and the vertex v to be located. It is required that some edge of the boundary of the Voronoi cell containing v in VD^* is a leaf in T^*. HandleCentroid returns the site s such that $v \in \mathsf{Vor}(s)$, and the additive distance to v. The algorithm is recursive, and bottoms out in one of two base cases (Line 7 or Line 11). The first way the recursion can end is if we reach the bottom of the centroid decomposition. If T^* is a singleton, its single node f^* corresponds to an edge in VD^* separating the Voronoi cells of two sites, say s_1 and s_2. At this point we know that either $v \in \mathsf{Vor}(s_1)$ or $v \in \mathsf{Vor}(s_2)$, and determine which case is true by comparing the additive distances from each of s_1 and s_2, which can be computed using the MSSP data structure (Lines 2– 7).

We next explain how to treat the case that T^* is not a singleton. The root f^* of T^* is dual to a trichromatic face f composed of three vertices y_0, y_1, y_2 in clockwise order, which are, respectively, in distinct Voronoi cells of sites s_0, s_1, s_2. Let e_0, e_1, e_2 be the edges y_2y_0, y_0y_1, y_1y_2, respectively. For $k \in \{0, 1, 2\}$, let p_k denote the s_k-to-y_k shortest path. Let C_k denote the path $p_k \circ e_k \circ \mathrm{rev}(p_{k-1 \pmod 3})$. A vertex of H either lies on one of the p_k's, or strictly to the right of exactly one of the C_k's. (The second case can be equivalently restated as follows: v is enclosed by the cycle comprised of C_k and the $s_{k-1 \pmod 3}$-to-s_k subpath of the face h that does not contain $s_{k+1 \pmod 3}$). See Figure **??**.

For each k, we can check whether v lies on some p_k using the MSSP data structure. If this is the case, then $v \in \mathsf{Vor}(s_k)$, and we are done (Lines 10– 11).

To support such queries with the MSSP data structure we need to augment it with additional decorations. For a shortest path tree F, rooted at a vertex of h, we define the preorder and postorder numbers of the the nodes of F by the order in which nodes are visited by a depth-first-search traversal of F in which darts incident to a node v are visited according to the permutation cycle in the embedding of H that corresponds to v, starting from the dart d whose tail is the parent of v in F (for the root of F, starting at an imaginary dart embedded in the face h). For a node v, preorder(v) (postorder(v)) is the number of darts of F traversed until the first (last) time v is visited by the traversal. Node x is an ancestor of node y if and only if preorder(x) $\leq$ preorder(y) $\leq$ *postorder*(y) $\leq$ *postorder*(x). For two nodes x and y, such that none is an ancestor of the other in F, we say that x is *right of* y in F if preorder(x) $<$ preorder(y). Otherwise, we say that x is *left of* y in F.

Lemma 12.3.4. *The MSSP data structure for a graph H with distinguished face h can be augmented to answer the following queries in $O(\log |H|)$ time per query.*

- *Return the distance in H from a node s on h to any node v of H.*
- *Return whether x is an ancestor of y in the shortest path tree rooted at a node s of h.*
- *Return whether x is right of y in the shortest path tree rooted at a node s of h.*

Problem 12.1. *Prove Lemma 12.3.4 by showing how to maintain preorder and postorder numbers as decorations of a link-cut tree representation of the primal shortest path tree during the execution of the MSSP algorithm.*

Lemma 12.3.5. *We can check whether v lies strictly to the right of C_k with a constant number of queries to an MSSP data structure for H with sources S.*

Proof. By Lemma 12.3.4 we can check which of the sites s_k and $s_{k-1 \pmod 3}$ is closer to v with respect to the additive distances with two queries to the MSSP data structure. Without loss of generality, suppose that s_k is closer to v.

We claim that v lies strictly to the right of C_k if and only if v is right of y_k^f in the shortest path tree rooted at s_k. This is because a shortest s_k-to-v path that emanates left of of the shortest s_k-to-y_k^f path must intersect $p_{k-1 \pmod 3}$. This is a contradiction since all vertices on $p_{k-1 \pmod 3}$ are in $\mathsf{Vor}(s_{k-1 \pmod 3})$.

Checking whether v is right of y_k^f in the shortest path tree rooted at s_k can also be done with a single query to the MSSP data structure by Lemma 12.3.4. □

When the algorithm finds that v is right of C_k, it recurses on T_k^*, the subtree of T^* rooted at the child of f^* that contains the leaf edge of VD^* representing e_k^* (Line 14).

Algorithm 12.1 HANDLECENTROID(T^*, v)

Input: A centroid decomposition tree T^* of a subtree of a Voronoi diagram VD^*, and the vertex v to be located.
Require: Some edge of the boundary of the Voronoi cell containing v in VD^* is a leaf in T^*.
Output: The site s such that $v \in \mathsf{Vor}(s)$, and the additive distance to v.

1: $f^* \leftarrow$ root of T^*
2: **if** T^* is a singleton **then**
3: $\quad s_1, s_2 \leftarrow$ sites corresponding to f^*
4: $\quad$ **for** $k = 1, 2$ **do**
5: $\quad\quad d_k \leftarrow \text{weight}(s_k) + d_H(s_k, v)$
6: $\quad j \leftarrow \text{argmin}_k(d_k)$
7: $\quad$ **return** (s_j, d_j)
8: $s_0, s_1, s_2 \leftarrow$ sites corresponding to f^*
9: **for** $k = 0, 1, 2$ **do**
10: $\quad$ **if** v lies on p_k **then** ▷ p_k is the s_k-to-y_k path in the shortest path tree of H rooted at s_k
11: $\quad\quad$ **return**$(s_k, \text{weight}(s_k) + d_H(s_k, v))$
12: $\quad$ **else if** v is (strictly) right of C_k **then** ▷ C_k is the concatenation of p_k, e_k, and reversed $p_{k-1 \pmod 3}$
13: $\quad\quad T_k^* \leftarrow$ subtree of T^* rooted at the child of f^* containing the leaf edge of VD^* representing e_k^*
14: $\quad\quad$ **return** HANDLECENTROID(T_k^*, v)

Lemma 12.3.6. HANDLECENTROID *is correct.*

Proof. Define $f, y_k, s_k, e_k^*, f^*, p_k, C_k$ as above, and let $\tilde{s}$ be such that $v \in \mathsf{Vor}(\tilde{s})$. If v is found to lie on p_k in Line 10, then $\tilde{s}$ is s_k, as returned in Line 11. The loop invariant is that T^* contains *some* leaf edge that belongs to the boundary of the cell $\mathsf{Vor}(\tilde{s})$. This is clearly true in the initial call, when T^* is the entire centroid decomposition of VD^*. Suppose that v is found to be strictly to the right of C_k in Line 12. Observe that since p_k and p_{k-1} are monochromatic, all edges of VD^* correspond to paths in H^* that are disjoint from the set of dual edges of C_k, with the exception of e_k^*. We claim that T_k^* contains at least one edge bounding $\mathsf{Vor}(\tilde{s})$. This is clearly true if e_k^* is such an edge, i.e. $\tilde{s} \in \{s_{k-1}, s_k\}$. In the complementary case, all vertices of $\mathsf{Vor}(\tilde{s})$ are strictly to the right of C_k. Hence, none of the edges bounding $\mathsf{Vor}(\tilde{s})$ can be in $T_{k'}^*$ for $k' \neq k$. Thus, the maintained invariant implies that there is such an edge in T_k^*.

When f^* is a single edge on the boundary of $\mathsf{Vor}(s_1), \mathsf{Vor}(s_2)$ the loop invariant guarantees that either $\tilde{s} = s_1$ or $\tilde{s} = s_2$. The additive distances d_1 and d_2 to s_1 and s_2 respectively are computed in Line 5, and $\tilde{s}$ is the site with smaller additive distance among the two (Line 7). Hence, Line 6 returns the correct answer. □

The efficiency of procedure HANDLECENTROID depends on the time required

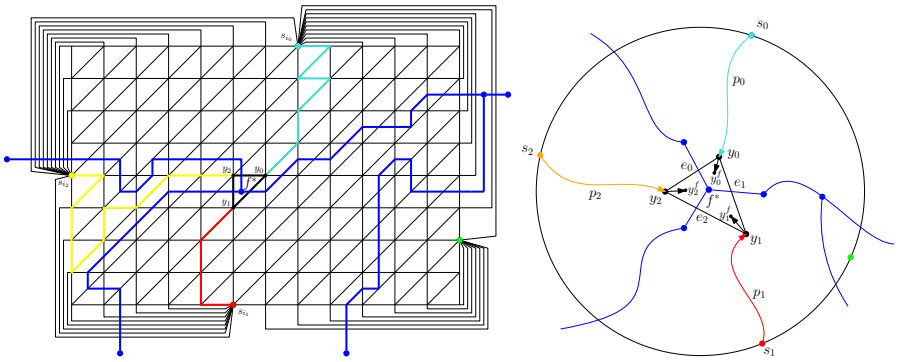

Figure 12.2: Illustration of the setting and proof of Lemma 12.3.6. Left: A decomposition of VD^* (shown in blue) by a centroid f^* into three subtrees, and a corresponding partition of P into three regions delimited by the paths p_i (shown in red, yellow, and turquoise). Right: a schematic illustration of the same scenario.

to compute distances in H (Lines 5 and 11) and the left/right/on relationship (Lines 10 and 12). By Lemma 12.3.5, given an MSSP data structure for H, with sources S, each of these operations can be performed in time $O(\log |H|)$ and hence Lemma 12.3.3 follows.

12.3.3 The oracle

For clarity of presentation, we first describe our oracle under the assumption that the boundary vertices of each piece P in the r-division of the graph lie on a single hole and that each such hole is a simple cycle. Multiple holes and non-simple cycles do not pose any significant complications; we explain how to treat pieces with multiple holes that are not necessarily simple cycles, separately. For a piece P of an r–division of a graph G, we denote by P^{out} the subgraph $G - (P - \partial P)$.

Data Structure. The data structure is recursive, with only 3 recursive levels. We compute an r-division with $r = n^{2/3}\sqrt{\log n}$. The data structure consists of the following for each piece P of the r-division:

1. If the recursive level is smaller than 3, the recursive data structure for P. If the recursive level is 3, a table storing for each pair of vertices u, v in P, the distance from u to v in P.

2. Two MSSP data structures, one for P and one for P^{out}, both with sources the nodes of ∂P. The MSSP data structure for P requires space $O(r \log r)$, while the one for P^{out} requires space $O(n \log n)$. The total space required for the MSSP data structures is $O(\frac{n^2}{r} \log n)$, since there are $O(\frac{n}{r})$ pieces.

3. For each node u of P:

   - $\mathsf{VD}^*_{in}(u, P)$, the dual representation of the Voronoi diagram for P with sites the nodes of ∂P, and additive weights the distances from u to these nodes in G;
   - $\mathsf{VD}^*_{out}(u, P)$, the dual representation of the Voronoi diagram for P^{out} with sites the nodes of ∂P, and additive weights the distances from u to these nodes in G.

   The representation of each Voronoi diagram occupies $O(\sqrt{r})$ space and hence, since each vertex belongs to a constant number of pieces, all Voronoi diagrams require space $O(n\sqrt{r})$.

The total space used by items 2,3 at each recursive level is $O(n^{4/3}\sqrt{\log n})$. In the third recursive level, since $r^3 = \tilde{O}(n^{8/27}) = O(n^{1/3})$, the total size of all tables in item 1 is $O(\frac{n}{r^3}(r^3)^2) = O(nr^3) = O(n^{4/3})$. Thus, the total space is $O(n^{4/3}\sqrt{\log n})$.

Query. We obtain a piece P of the r-division that contains u. Let us first suppose that $v \in P$. We have to consider both the case that the shortest u-to-v path crosses ∂P and the case that it does not. If it does cross, we retrieve this distance by performing a point location query for v in the Voronoi diagram $\mathsf{VD}_{in}(u, P)$. If the shortest u-to-v path does not cross ∂P, the path lies entirely within P. We thus retrieve the distance by querying the recursive distance oracle for P. The answer is the minimum of the two returned distances. Else, $v \notin P$ and the shortest path from u to v must cross ∂P. The answer can be thus obtained by a point location query for v in the Voronoi diagram $\mathsf{VD}_{out}(u, P)$ in time $O(\log^2 n)$ by Lemma 12.3.3. The pseudocode of the query algorithm is presented below as procedure SimpleDist(u, v) (Algorithm 12.2). Overall, we make at most one point location query at each recursive level, plus at most one table lookup in the third recursive level. Therefore the query time is $O(\log^2 n)$.

Algorithm 12.2 SimpleDist(u, v)

Input: Two nodes u and v.
Output: $d_G(u, v)$.

```
1: P ← a piece of the r-division containing u
2: if v ∈ P then
3:     d1 ← dP(u, v)
4:     d2 ← PointLocate(VD*in(u, P), v)
5:     return min(d1, d2)
6: else
7:     return PointLocate(VD*out(u, P), v)
```

Dealing with holes. The data structure has to be modified as follows.

2. For each hole h of P, two MSSP data structures, one for P and one for $P^{h,out}$, both with sources the nodes of ∂P that lie on h. Here, $P^{h,out}$ is the subgraph of P^{out} bounded by the hole h.

3. For each node u of P, for each hole h of P:

   - $\mathsf{VD}^*_{in}(u, P, h)$, the dual representation of the Voronoi diagram for P with sites the nodes of ∂P that lie on h, and additive weights the distances from u to these nodes in G;
   - $\mathsf{VD}^*_{out}(u, P, h)$, the dual representation of the Voronoi diagram for $P^{h,out}$ with sites the nodes of ∂P that lie on h, and additive weights the distances from u to these nodes in G.

As for the query, if $v \in P$ we have to perform a point location query in $\mathsf{VD}_{in}(u, P, h)$ for each hole h of P. Else $v \notin P$ and we have to perform a point location query in $\mathsf{VD}_{out}(u, P, h)$ for the hole h of P such that $v \in P^{h,out}$. We can afford to store the required information to identify this hole explicitly in balanced search trees.

We thus obtain the following result.

Theorem 12.3.7. *For a planar graph G of size n, there is an $O(n^{4/3}\sqrt{\log n})$-sized data structure that answers distance queries in time $O(\log^2 n)$.*