An Optimal Algorithm for L_1 Shortest Paths in Unit-Disk Graphs*

Haitao Wang† Yiming Zhao‡

Abstract

A unit-disk graph $G(P)$ of a set P of points in the plane is a graph with P as its vertex set such that two points of P are connected by an edge if the distance between the two points is at most 1 and the weight of the edge is equal to the distance of the two points. Given P and a source point $s \in P$, we consider the problem of finding shortest paths in $G(P)$ from s to all other vertices of $G(P)$. In the L_2 case where the distance is measured by the L_2 metric, the problem has been extensively studied and the current best algorithm runs in $O(n \log^2 n)$ time, with $n = |P|$. In this paper, we study the L_1 case in which the distance is measured under the L_1 metric (and each disk becomes a diamond); we present an $O(n \log n)$ time algorithm, which matches the $\Omega(n \log n)$-time lower bound.

1 Introduction

Let P be a set of n points in the plane. The *unit-disk graph* $G(P)$ of P is a graph with P as its vertex set such that two points of P are connected by an edge if the distance between the two points is at most 1. Alternatively, $G(P)$ is the intersection graph of the set of disks centered at the points of P with radii equal to 1/2. Each edge of $G(P)$ has a weight that is equal to the distance of the two incident vertices of the edge.

In this paper, we consider the *single-source shortest path* (SSSP) problem on $G(P)$, i.e., given P and a source point $s \in P$, compute shortest paths in $G(P)$ from s to all other points of P. In particular, we consider the L_1 case of the problem in which the distance is measured under the L_1 metric (and each disk becomes a diamond).

The L_2 case of the problem where the distance is measured under the L_2 metric has been extensively studied [3, 5, 8, 9, 15, 16]. The current best algorithm, which was given by Wang and Xue [16], runs in $O(n \log^2 n)$ time. The L_1 case, however, has not been particularly studied before. To solve the L_1 problem, we follow the algorithmic framework of Wang and Xue [16] but give a faster implementation. The runtime of Wang and Xue's algorithm [16] is dominated by a bottleneck subproblem. Due to some special properties of the L_1 metric, we derive a more efficient algorithm for the bottleneck subproblem in L_1 case, which leads to an overall $O(n \log n)$-time algorithm for the shortest path problem.

More specifically, the bottleneck subproblem is the offline insertion-only additively-weighted nearest-neighbor problem, where we are given an offline sequence of k insertions and queries such that an *insertion* inserts a weighted point to a point set U (which is $\emptyset$ initially) and a *query* asks for the additively-weighted nearest neighbor in U of a query point. The goal is to answer all queries. Wang and Xue [16] solved the problem in $O(k \log^2 k)$ time by using the standard logarithmic method [1, 2]. This leads to the overall $O(n \log^2 n)$ time for their shortest path algorithm [16]; reducing the time for the subproblem to $O(k \log k)$ would solve the shortest path problem in $O(n \log n)$ time. The difficulty in doing so is that there does not exist a semi-dynamic (for insertions only) weighted Voronoi diagram data structure that can perform each insertion in $O(\log k)$ amortized time (in order to answer queries, an efficient dynamic point location data structure is also needed). For solving our L_1 shortest path problem, we first observe that in the bottleneck subproblem U and V are separated by an axis-parallel line ℓ, where V is the set of all query points. Without loss of generality, we assume that ℓ is horizontal and U is below ℓ. Based on the properties of the L_1 metric, a critical observation we find is that the portion of the weighted L_1 Voronoi diagram of U above ℓ only consists of a set of vertical lines. Then, we can easily maintain these vertical lines by a balanced binary search tree so that each query can be answered in $O(\log k)$ time. Further, the special structure also allows us to update the portion of the Voronoi diagram above ℓ in $O(\log k)$ amortized time for each insertion. As such, the bottleneck subproblem can be solved in $O(k \log k)$ time in the L_1 case, which leads to an overall $O(n \log n)$ time algorithm for the shortest path problem. Note that the space of our shortest path algorithm is $O(n)$.

Cabello and Jejčič [3] observed that by a simple reduction from the max-gap problem, deciding whether the unit-disk graph $G(P)$ is connected requires $\Omega(n \log n)$ time even if all points of P are on a line. This implies that $\Omega(n \log n)$ is a lower bound for solving the shortest path problem in unit-disk graphs for both the L_1 and L_2 cases (because both cases are the same when all points

*This research was supported in part by NSF under Grant CCF-2005323.

†Department of Computer Science, Utah State University, Logan, UT 84322, USA. haitao.wang@usu.edu

‡Corresponding author. Department of Computer Science, Utah State University, Logan, UT 84322, USA. yiming.zhao@usu.edu

of P are on a line). As such, our algorithm for the L_1 case is optimal.

1.1 Related work

Before Wang and Xue's work [16], the shortest path problem in the L_2 case had been studied by many others. Roditty and Segal [15] gave the first sub-quadratic algorithm of $O(n^{4/3+\epsilon})$ time for any constant $\epsilon > 0$. Cabello and Jejčič [3] later proposed an improved algorithm of $O(n^{1+\epsilon})$ time. Following the framework of Cabello and Jejčič [3] but with a more efficient data structure for the bichromatic closest pair problem, Kaplan et al. [9] gave a randomized algorithm that solves the problem in $O(n \log^{12+o(1)} n)$ expected time. Approximation algorithms for the problem have also been developed, e.g., see [5, 8, 16]

The shortest path problem we consider is actually on a *weighted* unit-disk graph. In the *unweighted* case, the weight of each edge of the graph is 1. The unweighted problem is much easier. The L_2 unweighted problem can be solved in $O(n \log n)$ time [3, 5]. In particular, if all input points of P are presorted by their x- and y-coordinates, the algorithm of Chan and Skrepetos [4] runs in $O(n)$ time.

As an important class of geometric intersection graphs, unit-disk graphs have been widely studied due to many of their applications, e.g., in wireless sensor networks [13, 14]. In addition to the shortest path problem, many other problems on unit-disk graphs have also been considered in the literature, such as the clique problem [6], the independent set problem [12], all pairs of shortest paths [4, 5, 8], the diameter problem [4, 5, 8], etc. Comparing to general graphs, these problems in unit-disk graphs can be solved more efficiently by exploiting their underlying geometric structures.

Outline. In the following, we describe the main algorithm in Section 2 while the bottleneck subproblem is tackled in Section 3.

2 The main algorithm

In this section, we describe the main algorithm for the shortest path problem. Our algorithm follows Wang and Xue's algorithmic framework [16]. In the following, we will adapt their algorithm to the L_1 case. We will also borrow some of their notation.

For any two points p and q in the plane, we use $d(p, q)$ to denote their L_1 distance. For any point p, we use $\odot_p$ to denote the unit disk centered at p, which is a diamond in the L_1 metric. Let s be the source point of P. Throughout the paper, we will use the points of P and the vertices of the unit-disk graph $G(P)$ interchangeably.

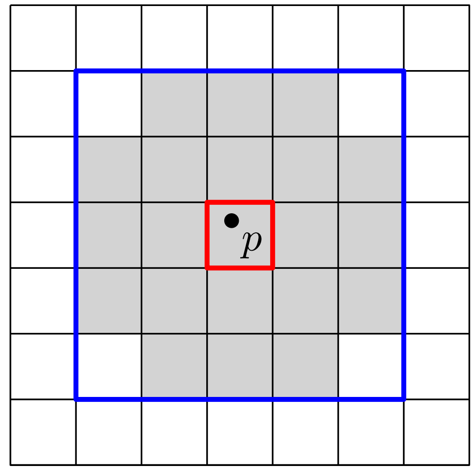

Figure 1: The side length of each square cell in the grid Γ is $\frac{1}{2}$. For the black point p, the red cell that contains it is $\square_p$, and the square area bounded by blue segments which contains 5×5 cells is the patch $\boxplus_p$. For any point in $\square_p$, its neighboring points in $G(P)$ must lie in the grey region.

The algorithm follows the basic idea of Dijkstra's shortest path algorithm with the help of a grid. At the outset, we implicitly build a grid Γ of square cells of side length 1/2. For simplicity of discussion, we assume that each vertex of $G(P)$ lies in the interior of a single cell of Γ. A *patch* of Γ is a square area consisting of 5×5 cells of Γ. For any point p in the plane, let $\square_p$ denote the cell of Γ that contains p and $\boxplus_p$ denote the patch whose central cell is $\square_p$ (e.g., see Fig. 1). Since the side length of each cell of Γ is 1/2, if two vertices of $G(P)$ are in a single cell of Γ, they must be connected by an edge in $G(P)$. On the other hand, if two points p and q are connected by an edge in $G(P)$, then q must be in a cell of $\boxplus_p$. Unlike Dijkstra's shortest path algorithm, which selects one single vertex in each iteration to compute shortest-path information, our algorithm tries to compute shortest-path information for all vertices in a cell of Γ and then pass shortest-path information to the vertices in the neighboring cells.

For a subset $Q \subseteq P$ and a cell $\square$ (resp., a patch $\boxplus$) of Γ, define $Q_\square = Q \cap \square$ (resp., $Q_\boxplus = Q \cap \boxplus$).

To implicitly compute the grid Γ, we actually perform the following preprocessing. We compute $P_\square$ for all cells $\square$ of Γ that contain at least one point of P. We also associate pointers to each point $p \in P$ such that from p we can access $\square_p$ and $\boxplus_p$. The preprocessing can be done in $O(n \log n)$ time and $O(n)$ space [16].

The algorithm will compute a table $dist[\cdot]$ for all vertices of $G(P)$, where $dist[p]$ is the length of a shortest path between s and a point $p \in P$. Note that we should also maintain the corresponding path-predecessor information to form a shortest path tree; this can be done by standard techniques [16], so we omit the discussions.

One important subroutine that will be extensively used in the algorithm is $\textsc{Update}(U, V)$. For two subsets $U, V \subseteq P$, $\textsc{Update}(U, V)$ is to update the shortest-path information of vertices in the set V by using the shortest-path information of vertices in U. More specifically, for each $v \in V$, let $q_v = \arg\min_{u \in U \cap \bigodot_v}\{dist[u] + d(u, v)\}$. The purpose of $\textsc{Update}(U, V)$ is to find q_v for all $v \in V$ and update $dist[v] = \min\{dist[v], dist[q_v] + d(q_v, v)\}$.

With $\textsc{Update}(U, V)$, the algorithm works as follows (refer to Algorithm 1 for the pseudocode). Initially, for each vertex $p \in P$, $dist[p]$ is set to ∞, except that $dist[s] = 0$. Initialize $Q = P$. In the main loop, as long as $Q \neq \emptyset$, in each iteration we find a vertex $q \in Q$ who has a minimum $dist[q]$. Subsequently there are two subroutines $\textsc{Update}(Q_{\boxplus_q}, Q_{\square_q})$ and $\textsc{Update}(Q_{\square_q}, Q_{\boxplus_q})$. Finally, vertices in $Q_{\square_q}$ are removed from Q, because $dist[p]$ for all $p \in Q_{\square_q}$ have been correctly computed. Refer to [16] for the correctness proof, which is applicable to the L_1 case.

Algorithm 1: The SSSP Algorithm [16]

```
Function SSSP(P, s):
    for each p ∈ P do
        dist[p] = ∞
    end
    dist[s] = 0
    Q = P
    while Q ≠ ∅ do
        q = arg min_{p∈Q} {dist[p]}
        Update(Q_⊞q, Q_□q) // first update
        Update(Q_□q, Q_⊞q) // second update
        Q = Q \ Q_□q
    end
    return dist[·]
end
```

Implementing the algorithm efficiently hinges on the two $\textsc{Update}$ procedures.

The first update. For the first update $\textsc{Update}(Q_{\boxplus_q}, Q_{\square_q})$, the key is to find a point $q_v \in Q_{\boxplus_q} \cap \bigodot_v$ that minimizes $dist[q_v] + d(q_v, v)$ for each point $v \in Q_{\square_q}$. If we assign each point in $Q_{\boxplus_q}$ a weight equal to its $dist$-value, then q_v is essentially the additively-weighted nearest neighbor of v in $Q_{\boxplus_q} \cap \bigodot_v$. To find q_v efficiently, a crucial observation found by Wang and Xue [16] (see Lemma 2.5 in [16], whose proof is applicable to the L_1 case) is that any point $p \in Q_{\boxplus_q}$ that minimizes $dist[p] + d(p, v)$ must be in $\bigodot_v$, i.e., the nearest neighbor of v in $Q_{\boxplus_q}$ is also the nearest neighbor of v in $Q_{\boxplus_q} \cap \bigodot_v$. Due to this observation, we can find q_v for all $v \in Q_{\square_q}$ as follows. First, we build an L_1 additively-weighted Voronoi diagram on vertices in $Q_{\boxplus_q}$ and then using the diagram to find the nearest neighbor for each $v \in Q_{\square_q}$. Constructing the diagram can be done in $O(|Q_{\boxplus_q}| \log |Q_{\boxplus_q}|)$ time and $O(|Q_{\boxplus_q}|)$ space (e.g., by using the abstract Voronoi diagram algorithm [11]), and all queries together take $O(|Q_{\square_q}| \log |Q_{\boxplus_q}|)$ time (e.g., build a point location data structure on the diagram in $O(|Q_{\boxplus_q}|)$ time [7, 10] and then perform point location queries for points of $Q_{\square_q}$, which take $O(\log |Q_{\boxplus_q}|)$ time each).

The second update. Implementing the second update $\textsc{Update}(Q_{\square_q}, Q_{\boxplus_q})$ is not that easy anymore because the above crucial observation does not hold. Since $Q_{\boxplus_q}$ has $O(1)$ cells of Γ, it suffices to perform $\textsc{Update}(Q_{\square_q}, Q_{\square})$ for all cells $\square \in \boxplus_q$.

If $\square$ is $\square_q$, then $Q_{\square_q} = Q_{\square}$. Since the distance between any two points in $\square_q$ is at most 1, we can use the following algorithm to implement $\textsc{Update}(Q_{\square_q}, Q_{\square})$. We first build an L_1 weighted Voronoi diagram on points of $Q_{\square_q}$ in $O(|Q_{\square_q}| \log |Q_{\square_q}|)$ time and $O(|Q_{\square_q}|)$ space [11], and then use it to find the weighted nearest neighbor q_v for each point $v \in Q_{\square_q}$. Clearly, the total time is $O(|Q_{\square_q}| \log |Q_{\square_q}|)$.

If $\square$ is not $\square_q$, then a critical property is that $\square$ and $\square_q$ are separated by an axis-parallel line ℓ. To perform $\textsc{Update}(Q_{\square_q}, Q_{\square})$, Wang and Xue [16] proposed the following approach (see Algorithm 2 for the pseudocode). Let $U = Q_{\square_q}$ and $V = Q_{\square}$. We first sort vertices in $U = \{u_1, u_2, ..., u_{|U|}\}$ by their $dist$-values such that $dist[u_1] \leq dist[u_2] \leq ... \leq dist[u_{|U|}]$. Then we partition V into subsets $V_i = \{v \in V \mid v \in \bigodot_{u_i}, v \notin \bigodot_{u_j} \text{ for all } j < i\}$, for all $i = 1, 2, \ldots, |U|$. For each $1 \leq i \leq |U|$, for each vertex $v \in V_i$, we find $q_v = \arg\min_{p \in U_i}\{dist[p] + d(p, v)\}$, where $U_i = \{u_i, u_{i+1}, \ldots, u_{|U|}\}$, and update $dist[v] = \min\{dist[v], dist[q_v] + d(q_v, v)\}$. This step is implemented by a for loop (Lines 6–13) in Algorithm 2. By the definition of V_i, we have $U \cap \bigodot_v \subseteq U_i$ for all $v \in V_i$. Also, Wang and Xue [16] proved that q_v found as above must be in $\bigodot_v$ (see Lemma 2.6 in [16], whose proof is applicable to the L_1 case). As such, $q_v = \arg\min_{p \in U \cap \bigodot_v}\{dist[p] + d(p, v)\}$. This proves the correctness of the algorithm.

We now analyze the runtime of the above algorithm. Sorting the vertices of U takes $O(|U| \log |U|)$ time. To compute the subsets V_i, $1 \leq i \leq |U|$, Wang and Xue [16] gave an algorithm of $O(k \log k)$ time (and $O(k)$ space) for the L_2 case (see Section 2.2.1 [16]) by making use of the property that U and V are separated by ℓ, where $k = |U| + |V|$. For the L_1 case, we can use the same algorithm; in fact, the algorithm becomes easier as a disk in the L_1 case is a diamond. We omit the details and conclude that the subsets V_i, $1 \leq i \leq |U|$, can be computed in $O(k \log k)$ time in the L_1 case. Next, the for loop

Algorithm 2: UPDATE(U, V) [16]

```
1  Function Update(U, V):
2      Sort(U = {u_1, u_2, ..., u_|U|}) // dist[u_1] ≤
           ... ≤ dist[u_|U|]
3      for i = 1, 2, ..., |U| do
4          V_i = {v ∈ V | v ∈ ⊙_{u_i}, v ∉
              ⊙_{u_j} for all j < i}
5      end
6      U' = ∅
7      for i = |U|, |U| − 1, ..., 1 do
8          U' = U' ∪ {u_i}
9          for each v ∈ V_i do
10             q_v = arg min_{u∈U'} {dist[u] + d(u, v)}
11             dist[v] =
                 min{dist[v], dist[q_v] + d(q_v, v)}
12         end
13     end
14 end
```

(Lines 6–13) is for the bottleneck subproblem mentioned in Section 1, i.e., the offline insertion-only additively-weighted nearest-neighbor problem. Indeed, if we assign each vertex in U a weight equal to its *dist*-value, then q_v is essentially the additively-weighted nearest neighbor of v in U', where $U' = U_i$ in the i-th iteration of the for loop. The set U' is dynamically changed with point insertions. Using the standard logarithmic method [1,2], Wang and Xue [16] solves the problem in $O(k \log^2 k)$ time. By exploring the properties of the L_1 metric, we give an $O(k \log k)$ time (and $O(k)$ space) algorithm in Section 3. As such, UPDATE$(Q_{\square_q}, Q_{\square})$ can be performed in $O(k \log k)$ time and $O(k)$ space, with $k = |Q_{\square_q}| + |Q_{\square}|$.

In summary, since $Q_{\boxplus_q}$ has $O(1)$ cells, the second update UPDATE$(Q_{\square_q}, Q_{\boxplus_q})$ can be implemented in $O(|Q_{\boxplus_q}| \log |Q_{\boxplus_q}|)$ time as $Q_{\square_q} \subseteq Q_{\boxplus_q}$. This leads to the following theorem.

Theorem 1 *Given a set P of n points in the L_1 plane and a source point $s \in P$, the shortest paths from s to all vertices in the unit-disk graph $G(P)$ can be computed in $O(n \log n)$ time and $O(n)$ space.*

Proof. As discussed before, constructing the grid Γ implicitly can be done in $O(n \log n)$ time and $O(n)$ space [16]. We have shown that both UPDATE procedures can be implemented in $O(|Q_{\boxplus_q}| \log |Q_{\boxplus_q}|)$ time and $O(|Q_{\boxplus_q}|)$ space. As such, each iteration of the while loop of Algorithm 1 can be implemented in $O(|Q_{\boxplus_q}| \log |Q_{\boxplus_q}|)$ time and $O(|Q_{\boxplus_q}|)$ space. As $\sum_{q \in Q} |Q_{\boxplus_q}| \leq 25n$, the total time of the algorithm is $O(n \log n)$. Note that the overall time of Line 8 and Line 11 of Algorithm 1 can be easily bounded by $O(n \log n)$ by using a balanced binary search tree. The total space of the algorithm is $O(n)$. $\square$

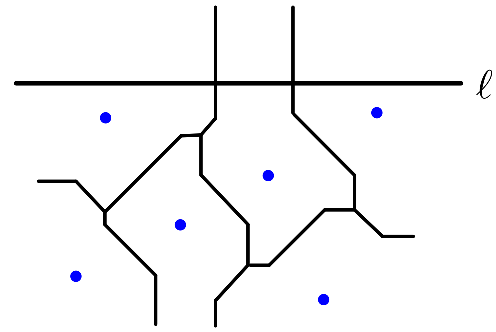

Figure 2: Illustrating $VD(U')$, where U' has six blue points (with the same weight). $VD_h(U')$ consists of two vertical half-lines.

3 The bottleneck subproblem

In this section, we present an $O(k \log k)$ time and $O(k)$ space algorithm to solve the bottleneck subproblem on U and V, with $k = |U| + |V|$. Recall U and V are separated by an axis-parallel line ℓ. Without loss of generality, we assume that ℓ is horizontal such that U is below ℓ and V is above ℓ. Our goal is to find $q_v \in U'$ for all $v \in V_i$ (i.e., Line 10 in Algorithm 2), for a subset $U' \subseteq U$.

In the following, we first discuss some observations about the geometric structure of the problem and then describe the algorithm.

3.1 Observations

Let $VD(U')$ denote the weighted Voronoi diagram of U'. To find q_v, it suffices to locate the cell of $VD(U')$ that contains v. Let h denote the upper half-plane bounded by ℓ. As v is above ℓ, it suffices to maintain the portion of $VD(U')$ above ℓ, denoted by $VD_h(U')$. In what follows, we first show that $VD_h(U')$ has a very simple structure: it only consists of a set of vertical half-lines with endpoints on ℓ and going upwards to the infinity (e.g., see Fig. 2). Then, we will show that $VD_h(U')$ can be updated in $O(\log k)$ amortized time for each insertion (i.e., inserting a point into U').

We say a vertical half-line is *grounded* on ℓ if it goes upwards to the infinity and has its endpoint on ℓ. For any point or a vertical line segment p in the plane, we use $x(p)$ to denote its x-coordinate. For each point $u \in U$, we define its weight $w(u) = dist[u]$.

Properties of bisectors of two weighted points. Consider two weighted points a and b in the plane with nonnegative weights $w(a)$ and $w(b)$, respectively. The *bisector* $B(a, b)$ of a and b is the locus of points with equal (additively-)weighted distance to a and b, i.e., $B(a,b) = \{p \in \mathbb{R}^2 \mid w(a) + d(a,p) = w(b) + d(b,p)\}$ (e.g., see Fig. 3). Note that in the degenerate case it is possible that an entire quadrant of the plane is in $B(a,b)$ (e.g., see Fig. 3b), in which case we only consider the vertical boundary of the quadrant to be in $B(a,b)$. Hence, $B(a,b)$ in general consists of three parts: two axis-parallel half-lines with a segment in the middle. Suppose both a and b are below the line ℓ and $x(a) \le x(b)$. Define $B_h(a,b) = B(a,b) \cap h$. Then either $B_h(a,b) = \emptyset$ or $B_h(a,b) \cap h$ is a vertical half-line grounded on ℓ; in the latter case $x(a) \le x(B_h(a,b)) \le x(b)$. Note that if $x(a) = x(b)$, then $B(a,b)$ is a horizontal line between a and b and thus $B_h(a,b) = \emptyset$.

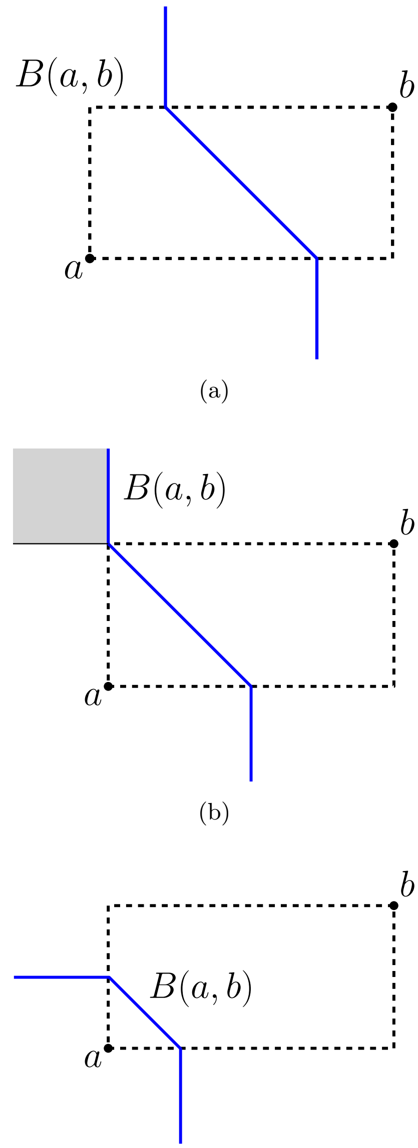

Figure 3: Possible cases for the bisector $B(a,b)$ of two weighted points a and b.

Geometric structure of $VD_h(U')$. Since all points of U are below ℓ, according to the discussion above, for any two points u_i and u_j of U, $B_h(u_i,u_j)$ is either $\emptyset$ or a vertical half-line grounded on ℓ (and the vertical half-line is between u_i and u_j). These properties guarantee that $VD_h(U')$ consists of a set of $O(|U'|)$ vertical half-lines grounded on ℓ (e.g., see Fig. 2), and between each pair of adjacent half-lines is the portion of the Voronoi cell of a vertex $u \in U'$. As such, we can use a balanced binary search tree $T(U')$ to store the x-coordinates of the vertical half-lines of $VD_h(U')$. Given a query point $v \in V$, we can use $T(U')$ to find the cell of $VD_h(U')$ containing v and thus obtain q_v in $O(\log |U'|)$ time, which is $O(\log k)$ as $|U'| \le |U| \le k$. In the following, we will discuss how to update $VD_h(U')$ after a point of U is inserted to U'. We first prove some properties about the geometric structure of $VD_h(U')$.

For each point $u \in U'$, let $R(u)$ denote the Voronoi cell of u in $VD(U')$ and let $R_h(u) = R(u) \cap h$. The above shows that if $R_h(u)$ is not empty, then it is bounded by two vertical half-lines from the left and right; let l_u and r_u denote these two half-lines, respectively. We call l_u the *left bounding half-line* and r_u the *right bounding half-line* of $R_h(u)$. Note that if $R_h(u)$ is the leftmost (resp., rightmost) cell of $VD_h(U')$, then we let l_u (resp., r_u) refer to the vertical half-line grounded on ℓ with x-coordinate $-\infty$ (resp., $+\infty$).

We say that a point $u \in U'$ is *relevant* if $R_h(u) \neq \emptyset$ and *irrelevant* otherwise. The following lemma proves several properties about the geometric structure of $VD_h(U')$, which will be useful for processing insertions.

Lemma 2 *Suppose $u^1, u^2, \ldots, u^t$ is the list of relevant vertices of U' whose Voronoi cells intersect h in the order from left to right. Then, the followings hold.*

1. *$x(u^1) < x(u^2) < \cdots < x(u^t)$.*
2. *For each $1 \le i < t$, r_{u^i} is $l_{u^{i+1}}$.*
3. *For each $1 \le i \le t$, $x(l_{u^i}) \le x(u^i) \le x(r_{u^i})$.*
4. *For each $1 \le i \le t$, p^i is in $R_h(u^i)$, where p^i is the vertical projection of u^i on ℓ.*

Proof. Consider a point u^i for any $i > 1$. By the definition of the list $u^1, u^2, \ldots, u^t$, l_{u^i} belongs to the bisector $B(u^{i-1}, u^i)$ of u^{i-1} and u^i, i.e., $l_{u^i} = B_h(u^{i-1}, u^i)$. According to the properties of bisectors, $x(u_{i-1}) \le x(l_{u^i}) \le x(u^i)$. Note that $x(u^{i-1}) = x(u^i)$ is not possible since otherwise $B_h(u^{i-1}, u^i)$ would be $\emptyset$ (contradicting with $l_{u^i} = B_h(u^{i-1}, u^i)$). As such, $x(u^{i-1}) < x(u^i)$ holds. This proves the first lemma statement.

According to our definition of the list $u^1, u^2, \ldots, u^t$, the left bounding half-line of $R_h(u^{i+1})$ must be the right bounding half-line of $R_h(u^i)$. Hence, the second lemma

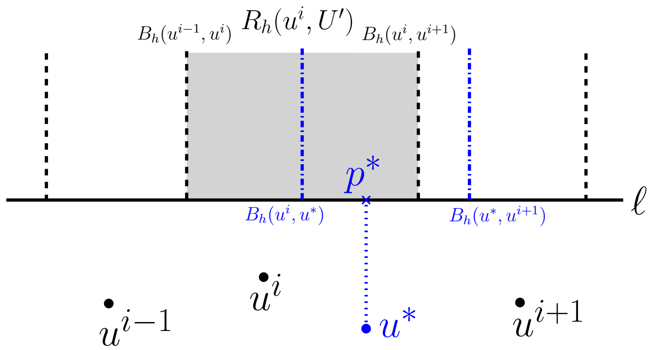

Figure 4: Illustrating $VD_h(U')$, and $VD_h(U'')$ after u^* is inserted. The two dash dotted blue segments are new half-lines in $VD_h(U'')$ while $B_h(u^i, u^{i+1})$ does not appear in $VD_h(U'')$. $R_h(u^i, U')$ is the grey area and $R_h(u^*, U'')$ is the region between the two dash dotted blue segments. Note that $B_h(u^{i-1}, u^i)$ is $l_{u^i} = r_{u^{i-1}}$ and $B_h(u^i, u^{i+1})$ is $r_{u^i} = l_{u^{i+1}}$.

statement holds.

The above shows that $x(l_{u^i}) \leq x(u^i)$ for $i > 1$. If $i = 1$, $x(l_{u^i}) \leq x(u^{i-1})$ also holds, for $x(l_{u^i}) = -\infty$. This proves that $x(l_{u^i}) \leq x(u^i)$ for any $1 \leq i \leq t$. By a symmetric analysis, we can show that $x(u^i) \leq x(r_{u^i})$ for any $1 \leq i \leq t$. This proves the third lemma statement.

The fourth lemma statement is an immediate consequence of the third lemma statement. $\square$

3.2 Processing insertions

We are now in a position to describe our algorithm for processing insertions.

Consider inserting a point $u^* \in U \setminus U'$ into U'. As $u^* \in U$, u^* is below ℓ. Let $U'' = U' \cup \{u^*\}$. Our goal is to construct $VD_h(U'')$ by modifying $VD_h(U')$, or more precisely, obtain the tree $T(U'')$ by modifying $T(U')$. For differentiation, for each vertex $u \in U''$, we use $R(u, U'')$ to denote the Voronoi cell of u in $VD(U'')$ and use $R(u, U')$ to denote the Voronoi cell of u in $VD(U')$. We define $R_h(u, U'')$ and $R_h(u, U')$ similarly. Let $u^1, u^2, \ldots, u^t$ be the list of relevant vertices of U' whose Voronoi cells intersect h ordered from left to right.

We first compute the vertical projection of u^* on ℓ and let p^* denote the projection point (e.g., see Fig. 4). Then, using the tree $T(U')$, we find the cell $R_h(u^i, U')$ of $VD_h(U')$ that contains p^*, for some relevant point $u^i \in U'$. For ease of discussion, we assume $1 < i < t$ and other cases can be handled similarly. The following lemma is obtained based on Lemma 2.

Lemma 3 *$R_h(u^*, U'') \neq \emptyset$ if and only if $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$, and if $R_h(u^*, U'') \neq \emptyset$, then $p^* \in R_h(u^*, U'')$.*

Proof. If $R_h(u^*, U'') \neq \emptyset$, then by Lemma 2, p^* must be in $R_h(u^*, U'')$ and this implies $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$ must hold. On the other hand, suppose $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$. Then, since $p^* \in R_h(u^i, U')$, $d(p^*, u^i) + w(u^i) \leq d(p^*, u) + w(u)$ holds for any vertex $u \in U'$. Therefore, $d(p^*, u) + w(u) \geq d(p^*, u^*) + w(u^*)$ holds for any $u \in U''$. This implies that u^* is the nearest neighbor of p^* in U''. As such, the point p^* must be in $R_h(u^*, U'')$ and $R_h(u^*, U'')$ cannot be empty. $\square$

With Lemma 3, our insertion algorithm proceeds as follows. We check whether $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$. If not, then $R_h(u^*, U'') = \emptyset$ by Lemma 3 and thus $VD_h(U'') = VD_h(U')$; hence, $T(U'') = T(U')$ and we are done with processing the insertion of u^*. In the following, we assume that $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$. By Lemma 3, $R_h(u^*, U'') \neq \emptyset$ and thus $VD_h(U'') \neq VD_h(U')$. Below we discuss how to modify $VD_h(U')$ to obtain $VD_h(U'')$.

For each vertex $u \in U'$, we still use l_u and r_u to denote the left and right bounding vertical half-lines of $R_h(u, U')$, respectively.

Since $p^* \in R_h(u^i, U')$, we have $x(u^*) = x(p^*) \in [x(l_{u^i}), x(r_{u^i})]$. By Lemma 2, $x(u^{i-1}) \leq x(r_{u^{i-1}}) = x(l_{u^i})$ and $x(r_{u^i}) = x(l_{u^{i+1}}) \leq x(u^{i+1})$. Therefore, $x(p^*) \in [x(u^{i-1}), x(u^{i+1})]$. Also by Lemma 2, $x(u^{i-1}) < x(u^i) < x(u^{i+1})$. Without loss of generality, we assume that $x(u^i) \leq x(p^*) < x(u^{i+1})$. We first discuss how to obtain the portion of $VD_h(U'')$ to the left of p^*. To this end, we consider the points $u^i, u^{i-1}, \ldots, u^1$ in this order.

First, for u^i, we compute the bisector $B(u^i, u^*)$ of u^i and u^*. Depending on whether $B_h(u^i, u^*) = B(u^i, u^*) \cap h$ is $\emptyset$, there are two cases.

- If $B_h(u^i, u^*) \neq \emptyset$, then $B_h(u^i, u^*)$ is a vertical half-line grounded on ℓ. Since $x(u^i) \leq x(u^*)$, according to the properties of bisectors, $x(u^i) \leq x(B_h(u^i, u^*)) \leq x(u^*)$. As $x(l_{u^i}) \leq x(u^i)$ and $x(u^*) \leq x(r_{u^i})$, $B_h(u^i, u^*)$ must be in the Voronoi cell $R_h(u^i, U')$ between l_{u^i} and p^* (e.g., see Fig. 4). Hence, $B_h(u^i, u^*)$ must be the right bounding half-line of the cell $R_h(u^i, U'')$ in $VD_h(U'')$ as well as the left bounding half-line of the cell $R_h(u^*, U'')$. We update the tree $T(U')$ accordingly (i.e., insert $B_h(u^i, u^*)$ to $T(U')$) and then halt the algorithm (i.e., the construction of $VD_h(U'')$ on the left of p^* is finished).

- If $B_h(u^i, u^*) = \emptyset$, then by our definition of bisectors (including our way for handling the degenerating case), since $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$, $d(p, u^i) + w(u^i) \geq d(p, u^*) + w(u^*)$ holds for any point $p \in h$. This implies that u^i is *dominated* by u^* with respect to the points of h, and thus

u^i becomes irrelevant in $VD_h(U'')$. As such, we remove l_{u^i} from $T(U')$. Note that l_{u^i} is $r_{u^{i-1}}$ by Lemma 3.

Next, we consider u^{i-1} in a way similar to the above for u^i. If $B_h(u^{i-1}, u^*) \neq \emptyset$, then $B_h(u^{i-1}, u^*)$ becomes the right bounding half-line of the cell $R_h(u^{i-1}, U'')$ in $VD_h(U'')$ as well as the left bounding half-line of $R_h(u^*, U'')$. We insert $B_h(u^{i-1}, u^*)$ into $T(U')$ and halt the algorithm. If $B_h(u^{i-1}, u^*) = \emptyset$, then since $p^* \in R_h(u^*, U'')$ by Lemma 3, $d(p^*, u^{i-1})+w(u^{i-1}) \geq d(p^*, u^*)+w(u^*)$. Further, by our definition of bisectors (including our way for handling the degenerating case), $d(p, u^{i-1})+w(u^{i-1}) \geq d(p, u^*)+w(u^*)$ holds for any point $p \in h$. Therefore, as above, u^{i-1} becomes irrelevant in $VD_h(U'')$. Accordingly, we remove $l_{u^{i-1}}$ from $T(U')$. We then proceed to considering u^{i-2} in the same way as above.

The above describes the algorithm for constructing $VD_h(U'')$ to the left of p^*. The algorithm for constructing $VD_h(U'')$ to the right of p^* is similar. One slight difference is that the algorithm starts with considering u^{i+1} instead of u^i by first removing r_{u^i} from $T(U')$. Then, we compute the bisector $B(u^*, u^{i+1})$. If $B_h(u^*, u^{i+1}) \neq \emptyset$, then $B_h(u^*, u^{i+1})$ becomes the right bounding half-line of $R_h(u^*, U'')$ as well as the left bounding half-line of $R_h(u^{i+1}, U'')$. We insert $B_h(u^*, u^{i+1})$ into $T(U')$ and halt the algorithm. If $B_h(u^*, u^{i+1}) = \emptyset$, then u^{i+1} becomes irrelevant and we proceed to considering u^{i+2} in the same way.

The above describes the algorithm for constructing $VD_h(U'')$ from $VD_h(U')$. The resulting tree $T(U')$ is $T(U'')$. The following lemma summarizes the time complexity of the insertion algorithm described above and proves the correctness of the algorithm.

Lemma 4 *After a point $u^* \in U$ is inserted into U', $VD_h(U'')$ can be computed from $VD_h(U')$ in $O((\delta + 1)\log k)$ time, where $U'' = U' \cup \{u^*\}$ and δ is the number of relevant vertices of $VD_h(U')$ that become irrelevant in $VD_h(U'')$.*

Proof. The runtime of the insertion algorithm is obvious from our algorithm description. In the following, we prove the correctness of the algorithm.

If $d(p^*, u^i) + w(u^i) < d(p^*, u^*) + w(u^*)$, then $VD_h(U'') = VD_h(U')$ by Lemma 3 and thus our algorithm is correct in this case. In the following, we assume that $d(p^*, u^i)+w(u^i) \geq d(p^*, u^*)+w(u^*)$ and prove that the diagram $VD_h(U'')$ constructed by our algorithm is correct.

Let p be any point in h and let u be the point of U'' such that p is in the cell of u after our insertion algorithm for u^* is finished, i.e., $p \in R_h(u, U'')$. To prove the correctness of our algorithm, it suffices to show

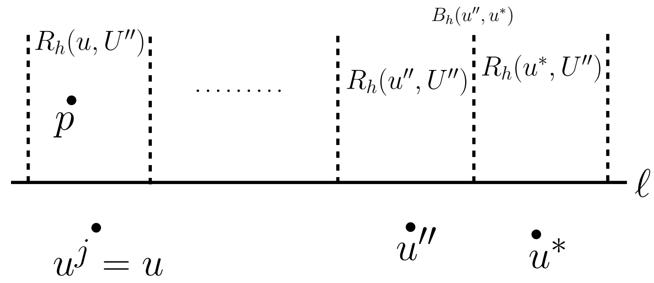

Figure 5: Illustrating the proof of Lemma 4 for the case where u is not adjacent to u^* in L.

that $d(p, u) + w(u) \leq d(p, u') + w(u')$ holds for every point $u' \in U''$. Depending on whether $u = u^*$, there are two cases. Let u^j be the point of U' such that $p \in R_h(u^j, U')$.

- We first consider the case $u = u^*$. As $p \in R_h(u^j, U')$, $d(p, u^j)+w(u^j) \leq d(p, u')+w(u')$ holds for any $u' \in U'$. As p is in the cell of u^* after the insertion algorithm finishes, according to our algorithm, $d(p, u^*) + w(u^*) \leq d(p, u^j) + w(u^j)$ must hold. Since $u = u^*$, we obtain that $d(p, u)+w(u) = d(p, u^*)+w(u^*) \leq d(p, u^j)+w(u^j) \leq d(p, u')+w(u')$ holds for any $u' \in U''$.

- We then consider the case $u \neq u^*$. In this case, according to our algorithm, u must be u^j and u and u^* define different cells in $VD_h(U'')$, i.e., $R_h(u, U'') \neq R_h(u^*, U'')$. Without loss of generality, we assume that $R_h(u, U'')$ is to the left of $R_h(u^*, U'')$. Depending on whether u is adjacent to u^* in the relevant point list L after the insertion algorithm (L is defined in the same way as Lemma 2 with respect to $VD_h(U'')$), there are two subcases.

  If u is adjacent to u^* in L, then since p is in the cell of u after the insertion algorithm, it holds that $d(p, u)+w(u) \leq d(p, u^*)+w(u^*)$. Since $u = u^j$ and $d(p, u^j)+w(u^j) \leq d(p, u')+w(u')$ holds for any $u' \in U'$, we obtain that $d(p, u) + w(u) \leq d(p, u') + w(u')$ holds for any $u' \in U''$.

  If u is not adjacent to u^* in L, then let u'' be the left neighboring relevant point of u^* in L (e.g., see Fig 5). Since $R_h(u, U'')$ is to the left of $R_h(u^*, U'')$ and $p \in R_h(u, U'')$, p must be to the left of $B_h(u'', u^*)$, which is the right bounding half-line of $R_h(u'', U'')$. As u'' is the left neighboring relevant point of u^* in L, according to our insertion algorithm, $d(p', u'') + w(u'') \leq d(p', u^*) + w(u^*)$ for any point $p' \in h$ to the left of $B_h(u'', u^*)$. Because p is in h to the left of $B_h(u'', u^*)$, $d(p, u'') + w(u'') \leq d(p, u^*) + w(u^*)$ holds. As $d(p, u^j) + w(u^j) \leq d(p, u') + w(u')$ for any $u' \in U'$, we have $d(p, u^j)+w(u^j) \leq d(p, u'')+w(u'')$. We thus derive $d(p, u^j) + w(u^j) \leq d(p, u^*) + w(u^*)$. Since $u = u^j$, we obtain that $d(p, u) + w(u) \leq d(p, u') + w(u')$ for any $u' \in U''$.

In summary, $d(p, u) + w(u) \leq d(p, u') + w(u')$ holds for every point $u' \in U''$. This proves the correctness of our algorithm. $\square$

Note that once a relevant point becomes irrelevant after an insertion, it will never become relevant again for any insertions in future. Therefore, the total sum of δ in Lemma 4 for processing all insertions of U is at most k. As such, by Lemma 4, the total time for processing all insertions is $O(k \log k)$.

Recall that all query operations can be performed in overall $O(k \log k)$ time by using the tree $T(U')$. Note that the space of our algorithm is bounded by $O(k)$. Therefore, we finally obtain the following result.

Lemma 5 *The bottleneck subproblem on U and V can be solved in $O(k \log k)$ time and $O(k)$ space, where $k = |U| + |V|$.*

References

[1] J.L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8:244–251, 1979.

[2] M. de Berg, K. Buchin, B.M.P. Jansen, and G. Woeginger. Fine-grained complexity analysis of two classic TSP variants. *ACM Transactions on Algorithms*, 17(1):5:1–5:29, 2021.

[3] S. Cabello and M. Jejčič. Shortest paths in intersection graphs of unit disks. *Computational Geometry: Theory and Applications*, 48(4):360–367, 2015.

[4] T.M. Chan and D. Skrepetos. All-pairs shortest paths in unit-disk graphs in slightly subquadratic time. In *Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC)*, pages 24:1–24:13, 2016.

[5] T.M. Chan and D. Skrepetos. Approximate shortest paths and distance oracles in weighted unit-disk graphs. In *Proceedings of the 34th International Symposium on Computational Geometry (SoCG)*, pages 24:1–24:13, 2018.

[6] B.N. Clark, C.J. Colbourn, and D.S. Johnson. Unit disk graphs. *Discrete mathematics*, 86(1-3):165–177, 1990.

[7] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[8] J. Gao and L. Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. *SIAM Journal on Computing*, 35(1):151–169, 2005.

[9] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2495–2504, 2017.

[10] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[11] R. Klein. Concrete and abstract Voronoi diagrams. volume 400 of *Lecture Notes in Computer Science*, Springer-Verlag, 1989.

[12] T. Matsui. Approximation algorithms for maximum independent set problems and fractional coloring problems on unit disk graphs. In *Japanese Conference on Discrete and Computational Geometry*, pages 194–200, 1998.

[13] C.E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, pages 234–244, 1994.

[14] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100, 1999.

[15] L. Roditty and M. Segal. On bounded leg shortest paths problems. *Algorithmica*, 59(4):583–600, 2011.

[16] H. Wang and J. Xue. Near-optimal algorithms for shortest paths in weighted unit-disk graphs. *Discrete and Computational Geometry*, 64:1141–1166, 2020.