Near-Optimal Algorithms for Shortest Paths in Weighted Unit-Disk Graphs*

Haitao Wang
Department of Computer Science, Utah State University, Logan, UT 84322, USA
https://cs.usu.edu/people/haitaowang
haitao.wang@usu.edu

Jie Xue
Department of Computer Science and Engineering, University of Minnesota – Twin Cities, Minneapolis, MN 55455, USA
http://cs.umn.edu/~xuexx193
xuexx193@umn.edu

Abstract

We revisit a classical graph-theoretic problem, the *single-source shortest-path* (SSSP) problem, in weighted unit-disk graphs. We first propose an exact (and deterministic) algorithm which solves the problem in $O(n \log^2 n)$ time using linear space, where n is the number of the vertices of the graph. This significantly improves the previous deterministic algorithm by Cabello and Jejčič [CGTA'15] which uses $O(n^{1+\delta})$ time and $O(n^{1+\delta})$ space (for any small constant $\delta > 0$) and the previous randomized algorithm by Kaplan et al. [SODA'17] which uses $O(n \log^{12+o(1)} n)$ expected time and $O(n \log^3 n)$ space. More specifically, we show that if the 2D offline insertion-only (additively-)weighted nearest-neighbor problem with k operations (i.e., insertions and queries) can be solved in $f(k)$ time, then the SSSP problem in weighted unit-disk graphs can be solved in $O(n \log n + f(n))$ time. Using the same framework with some new ideas, we also obtain a $(1+\varepsilon)$-approximate algorithm for the problem, using $O(n \log n + n \log^2(1/\varepsilon))$ time and linear space. This improves the previous $(1+\varepsilon)$-approximate algorithm by Chan and Skrepetos [SoCG'18] which uses $O((1/\varepsilon)^2 n \log n)$ time and $O((1/\varepsilon)^2 n)$ space. Because of the $\Omega(n \log n)$-time lower bound of the problem (even when approximation is allowed), both of our algorithms are almost optimal.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Single-source shortest paths, Weighted unit-disk graphs, Geometric graph algorithms

Digital Object Identifier 10.4230/LIPIcs.SoCG.2019.60

Related Version A full version of this paper is available at https://arxiv.org/abs/1903.05255.

Funding The research of Jie Xue is supported, in part, by a Doctoral Dissertation Fellowship from the Graduate School of the University of Minnesota.

Acknowledgements The authors would like to thank Timothy Chan for the discussion, and in particular, for suggesting the algorithm for Lemma 7. Jie Xue would like to thank his advisor Ravi Janardan for his consistent advice and support.

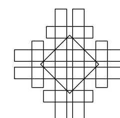
1 Introduction

Given a set S of n points in the plane, its *unit-disk graph* is an undirected graph in which the vertices are points of S and two vertices are connected by an edge iff the (Euclidean) distance between them is at most 1. Unit-disk graphs can be viewed as the intersection graphs of equal-sized disks in the plane, and find many applications such as modeling the topology of ad-hoc communication networks. As an important class of geometric intersection graphs,

* The work was partially done when Jie Xue was visiting Utah State University.

© Haitao Wang and Jie Xue;
licensed under Creative Commons License CC-BY

35th International Symposium on Computational Geometry (SoCG 2019).
Editors: Gill Barequet and Yusu Wang; Article No. 60; pp. 60:1–60:13
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



unit-disk graphs have been extensively studied in computational geometry. Many problems that are difficult in general graphs have been efficiently solved (exactly or approximately) in unit-disk graphs by exploiting their underlying geometric structures.

In this paper, we consider a classical graph-theoretic problem, the *single-source shortest-path* (SSSP) problem, in unit-disk graphs. Given an edge-weighted graph $G = (V, E)$ and a source vertex $s \in V$, the SSSP problem aims to compute shortest paths from s to all other vertices in G (or equivalently a shortest-path tree from s). In unit-disk graphs, there are two natural ways to weight the edges. The first way is to equally weight all the edge (usually called *unweighted* unit-disk graphs), while the second way is to assign each edge (a, b) a weight equal to the (Euclidean) distance between a and b (usually called *weighted* unit-disk graphs). The SSSP problem in a general graph has a trivial $\Omega(|E|)$-time lower bound, because specifying the edges of the graph already takes $\Omega(|E|)$ time. However, this lower bound does not hold in unit-disk graphs. A unit-disk graph (either unweighted or weighted), though having quadratic number of edges in worst case (e.g., all the vertices are very close to each other), can be represented by only giving the locations of its vertices in the plane. This linear-complexity representation allows us to solve the SSSP problem without explicitly constructing the graph and hence beat the $\Omega(|E|)$-time lower bound.

In unweighted unit-disk graphs, the SSSP problem is relatively easy, and various algorithms are known for solving it *optimally* in $O(n \log n)$ time [2, 3]. However, the weighted case is much more challenging. Despite of much effort made over years [2, 5, 9, 10, 12], state-of-the-art algorithms are still far away from being optimal. In this paper, we present new exact and approximation algorithms for the problem in weighted unit-disk graphs, which significantly improve the previous results and almost match the lower bound of the problem.

Organization. The remaining paper is organized as follows. In Sect. 1.1, we discuss the related work and our contributions. Sect. 1.2 presents some notations used throughout the paper. Our exact and approximation algorithms are given in Sect. 2 and 3, respectively. Due to the page limit, some lemma proofs are omitted but can be found in the full version [13].

1.1 Related work and our contributions

Besides the SSSP problem, many graph-theoretic problems have also been studied in unit-disk graphs, such as maximum independent set [11], maximum clique [6], distance oracle [5, 9], diameter computing [5, 9], all-pair shortest paths [3, 4], etc. Most of these problems have much more efficient solutions in unit-disk graphs than in general graphs.

The SSSP problem in unit-disk graphs has received a considerable attention in the last decades. The problem has an $\Omega(n \log n)$-time lower bound even when approximation is allowed, because deciding the connectivity of a unit-disk graph requires $\Omega(n \log n)$ time [2]. In unweighted unit-disk graphs, at least two $O(n \log n)$-time SSSP algorithms were known [2, 3], which are optimal. If the vertices are pre-sorted by their x- and y-coordinates, the algorithm in [3] can solve the problem in $O(n)$ time. In weighted unit-disk graphs, the SSSP problem was studied in [2, 5, 9, 10, 12]. Both exact and approximation algorithms were given to solve the problem in sub-quadratic time. For the exact case, the best known results are the deterministic algorithm by Cabello and Jejčič [2] which uses $O(n^{1+\delta})$ time and $O(n^{1+\delta})$ space (for any small constant $\delta > 0$) and the randomized algorithm by Kaplan et al. [10] which uses $O(n \log^{12+o(1)} n)$ expected time and $O(n \log^3 n)$ space. For the approximation case, the best known result is the $(1 + \varepsilon)$-approximate algorithm by Chan and Skrepetos [5] which uses $O((1/\varepsilon)^2 n \log n)$ time and $O((1/\varepsilon)^2 n)$ space.

In this paper, we first propose an exact SSSP algorithm in weighted unit-disk graphs which uses $O(n \log^2 n)$ time and $O(n)$ space, significantly improving the results in [2, 10]. Using the same framework together with some new ideas, we also obtain a $(1+\varepsilon)$-approximate algorithm which uses $O(n \log n + n \log^2(1/\varepsilon))$ time and $O(n)$ space, improving the result in [5]. Table 1 presents the comparison of our new algorithms with the previous results.

Table 1 Summary of the previous and our new algorithms for SSSP in weighted unit-disk graphs.

Type	Source	Time	Space	Rand./Det.
Exact	[12]	$O(n^{4/3+\delta})$	$O(n^{1+\delta})$	Deterministic
	[2]	$O(n^{1+\delta})$	$O(n^{1+\delta})$	Deterministic
	[10]	$O(n \log^{12+o(1)} n)$	$O(n \log^3 n)$	Randomized
	Corollary 9	$O(n \log^2 n)$	$O(n)$	Deterministic
Approximate	[9]	$O((1/\varepsilon)^3 n^{1.5} \sqrt{\log n})$	$O((1/\varepsilon)^4 n \log n)$	Deterministic
	[5]	$O((1/\varepsilon)^2 n \log n)$	$O((1/\varepsilon)^2 n)$	Deterministic
	Corollary 15	$O(n \log n + n \log^2(1/\varepsilon))$	$O(n)$	Deterministic

More specifically, our algorithms solve the SSSP problem in weighted unit-disk graphs by reducing it to the (2D) offline insertion-only additively-weighted nearest-neighbor (OIWNN) problem, in which we are given a sequence of operations each of which is either an insertion (inserting a weighted point in $\mathbb{R}^2$ to the dataset) or a weighted nearest-neighbor query (asking for the additively-weighted nearest neighbor of a given query point in the dataset) and our goal is to answer all the queries. The reductions imply the following results.

- If the OIWNN problem with k operations can be solved in $f(k)$ time, then the exact SSSP problem in weighted unit-disk graphs can be solved in $O(n \log n + f(n))$ time.
- If the OIWNN problem with k_1 operations in which at most k_2 operations are insertions can be solved in $f(k_1, k_2)$ time, then the $(1+\varepsilon)$-approximate SSSP problem in weighted unit-disk graphs can be solved in $O(n \log n + n \log(1/\varepsilon) + f(n, O(\varepsilon^{-2})))$ time.

Our time bounds in Table 1 are derived from the above results by arguing that $f(k) = O(k \log^2 k)$ and $f(k_1, k_2) = O(k_1 \log^2 k_2)$. Therefore, the bottleneck of our algorithms in fact comes from the OIWNN problem.

As an immediate application, our approximation algorithm can be applied to improve the preprocessing time of the distance oracles in weighted unit-disk graphs given by Chan and Skrepetos [5] (see the full version [13] for the details).

1.2 Notations

Basic notations. Throughout the paper, the notation $\|\cdot\|$ denotes the Euclidean norm; therefore, for two points $a, b \in \mathbb{R}^2$, $\|a - b\|$ is the Euclidean distance between a and b. For a point $a \in \mathbb{R}^2$, we use $\odot_a$ to denote the unit disk (i.e., disk of radius 1) centered at a.

Graphs. Let $G = (V, E)$ be an edge-weighted undirected graph. A path in G is represented as a sequence $\pi = \langle z_1, \dots, z_t \rangle$ where $z_1, \dots, z_t \in V$ and $(z_i, z_{i+1}) \in E$ for all $i \in \{1, \dots, t-1\}$; the *length* of π is the sum of the weights of the edges $(z_1, z_2), \dots, (z_{t-1}, z_t)$. For two vertices $u, v \in V$, we use $\pi_G(u, v)$ to denote the shortest path from u to v in G and use $d_G(u, v)$ to denote the length of $\pi_G(u, v)$. We say $v' \in V$ is the u-predecessor of v if $(v', v) \in E$ is the last edge of $\pi_G(u, v)$. For two paths π and π' in G where π is from u to v and π' is from v to w, we denote by $\pi \circ \pi'$ the *concatenation* of π and π', which is a path from u to w in G.

2 The exact algorithm

In this section, we describe our exact algorithm. Given a set S of n points in the plane and a source $s \in S$, our goal is to compute a shortest-path tree from s in the weighted unit-disk graph G induced by S. For all $a \in S$, we use $\lambda_a \in S$ to denote the s-predecessor of a. Specifically, we aim to compute two tables dist[$\cdot$] and pred[$\cdot$] indexed by the points in S, where $\text{dist}[a] = d_G(s,a)$ and $\text{pred}[a] = \lambda_a$.

We first briefly review how the well-known Dijkstra's algorithm computes shortest paths from a source s in a graph G. Initially, the algorithm sets all dist-values to infinity except $\text{dist}[s] = 0$, and sets $A = S$. Then it keeps doing the following procedure until $A = \emptyset$.

1. Pick the vertex $c \in A$ with the smallest dist-value.
2. For all $b \in A$ that are neighbors of c, update the value dist[b] using c, i.e., $\text{dist}[b] \leftarrow \min\{\text{dist}[b], \text{dist}[c] + w(c,b)\}$, where $w(c,b)$ is the weight of the edge (c,b).
3. Remove c from A.

Directly applying Dijkstra's algorithm to solve the SSSP problem in a weighted unit-disk graph takes quadratic time, since the graph can have $\Omega(n^2)$ edges in worst-case.

Our algorithm will follow the spirit of Dijkstra's algorithm in a high level and exploit many insights of unit-disk graphs in order to achieve a near-linear running time. First of all, we (implicitly) build a grid Γ on the plane, which consists of square cells with side-length $1/2$ (a similar grid is also used in [3]). Assume for convenience that no point in S lies on a grid line, and hence each point in S is contained in exactly one cell of Γ. A *patch* of Γ is a square area consisting of 5×5 cells of Γ. For a point $a \in S$, let $\square_a$ denote the cell of Γ containing a and $\boxplus_a$ denote the patch of Γ whose central cell is $\square_a$. For a set P of points in $\mathbb{R}^2$ and a cell $\square$ (resp., a patch $\boxplus$) of Γ, define $P_\square = P \cap \square$ (resp., $P_\boxplus = P \cap \boxplus$). We notice the following simple fact.

▶ **Fact 1.** *For all $a \in S$, we have $S_{\square_a} \subseteq \mathsf{NB}_G(a) \subseteq S_{\boxplus_a}$, where $\mathsf{NB}_G(a)$ is the set of all neighbors of a in G.*

We compute and store $S_\square$ (resp., $S_\boxplus$) for all cells $\square$ (resp., patches $\boxplus$) of Γ that contain at least one point in S. In addition, we associate pointers to each $a \in S$ so that from a one can get access to the stored sets $S_{\square_a}$ and $S_{\boxplus_a}$. The above preprocessing can be easily done in $O(n \log n)$ time and $O(n)$ space after computing $\square_a$ for all $a \in S$. We give in the full version a method to compute $\square_a$ for all $a \in S$ in $O(n \log n)$ time without using the floor function.

In order to present our algorithm, we first define a sub-routine Update as follows. Suppose we are now at some point of the algorithm. If U and V are two subsets of S, then the procedure Update(U,V) conceptually does the following.

1. $\text{dist}'[u] \leftarrow \text{dist}[u]$ for all $u \in U$.
2. $p_v \leftarrow \arg\min_{u \in U \cap \odot_v} \{\text{dist}'[u] + \|u - v\|\}$ for all $v \in V$.
3. For all $v \in V$, if $\text{dist}[v] > \text{dist}'[p_v] + \|p_v - v\|$, then update $\text{dist}[v] \leftarrow \text{dist}'[p_v] + \|p_v - v\|$ and $\text{pred}[v] \leftarrow p_v$.

In words, Update(U,V) updates the shortest-path information of the points in V using the shortest-path information of the points in U. We use lazy update by copying the dist[$\cdot$] table to dist$'$[$\cdot$] to guarantee that the order we consider the points in V does not influence the result of the update (note that U and V may not be disjoint). However, when U and V are not disjoint, lazy update may result in an inconsistency of shortest-path information, i.e., $\text{dist}[v] > \text{dist}[\text{pred}[v]] + \|\text{pred}[v] - v\|$ for some $v \in V$ after Update(U,V). This can happen when $p_v \in U \cap V$: for example, we update dist[v] to $\text{dist}'[p_v] + \|p_v - v\|$ and at the same

time $\text{dist}[p_v]$ also gets updated (hence $\text{dist}[p_v] < \text{dist}'[p_v]$), then $\text{dist}[v] > \text{dist}[p_v] + \|p_v - v\|$ after $\textsc{Update}(U, V)$. We call such a phenomenon *data inconsistency.* Although $\textsc{Update}$ can result in data inconsistency in general, we shall guarantee it never happens in our algorithm.

The main framework of our algorithm is quite simple, which is presented in Algorithm 1. Similarly to Dijkstra's algorithm, we also maintain a subset $A \subseteq S$ during the algorithm and pick the point $c \in A$ with the smallest dist-value in each iteration (line 6). The difference is that, instead of using c to update (the shortest-path information of) its neighbors, our algorithm tries to use all points in $A_{\square_c}$ to update their neighbors (line 8) and then remove them simultaneously from A (line 9). However, it is not guaranteed that the shortest-path information of all the points in $A_{\square_c}$ is correct when c is picked. Therefore, before using the points in $A_{\square_c}$ to update their neighbors, we use an extra procedure to "correct" the shortest-path information of these points, which is not needed in Dijkstra's algorithm. Surprisingly, we achieve this by simply updating the points in $A_{\square_c}$ once using the current shortest-path information of their neighbors (line 7).

Algorithm 1 SSSP(S, s).

```
1:  dist[a] ← ∞ for all a ∈ S
2:  pred[a] ← NIL for all a ∈ S
3:  dist[s] ← 0
4:  A ← S
5:  while A ≠ ∅ do                                    ▷ Main loop
6:      c ← arg min_{a∈A}{dist[a]}
7:      UPDATE(A_⊞c, A_□c)                            ▷ First update
8:      UPDATE(A_□c, A_⊞c)                            ▷ Second update
9:      A ← A\A_□c
10: return dist[·] and pred[·]
```

The correctness of our algorithm is non-obvious. Suppose m is the number of the iterations in the main loop. Let c_i be the point c picked in the i-th iteration.

▶ **Fact 2.** *The points $c_1, \dots, c_m$ belong to different cells in Γ.*

To prove the algorithm correctness, we first show that the dist-values of all points in S are correctly computed eventually. Clearly, during the entire algorithm, the dist-values can only decrease and never become smaller than the true shortest-path distances, i.e., we always have $\text{dist}[a] \geq d_G(s, a)$ for all $a \in S$. Keeping this in mind, we prove the following lemma.

▶ **Lemma 3.** *Algorithm 1 has the following properties.*
(1) *When the i-th iteration begins,* $\text{dist}[a] = d_G(s, a)$ *for all $a \in S$ with $d_G(s, a) \leq d_G(s, c_i)$.*
(2) *After the first update of the i-th iteration,* $\text{dist}[a] = d_G(s, a)$ *for all $a \in S_{\square_{c_i}}$.*
(3) *When the i-th iteration ends,* $\text{dist}[a] = d_G(s, a)$ *for all $a \in S$ with $\lambda_a \in S_{\square_{c_i}}$.*

Proof. We first notice that the property **(3)** follows immediately from the property **(2)** due to the second update. Indeed, for a point $a \in S$, if $\lambda_a \in S_{\square_{c_i}}$, then $a \in S_{\boxplus_{c_i}}$. If $a \in A_{\boxplus_{c_i}}$, then the property **(2)** implies that the second update makes $\text{dist}[a] = d_G(s, a)$. If $a \in S_{\boxplus_{c_i}} \backslash A_{\boxplus_{c_i}}$, then $a \in A_{\square_{c_j}}$ for some $j < i$ (since a got removed from A in a previous iteration) and the property **(2)** guarantees that $\text{dist}[a] = d_G(s, a)$ after the first update of the j-th iteration. As such, we only need to verify the first two properties. We achieve this using induction on i. The base case is $i = 1$. Note that $c_1 = s$ and $d_G(s, c_1) = 0$. Thus, to see **(1)**, we only need to guarantee that $\text{dist}[s] = d_G(s, s) = 0$ when the first iteration begins, which is

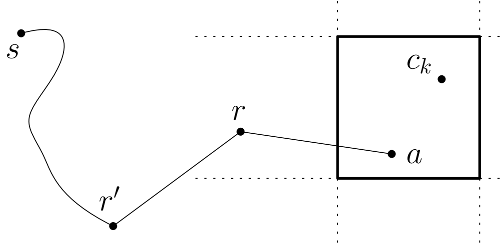


Figure 1 Illustrating points a, r, and r'. The solid path is $\pi_G(s,a)$. The solid square is $\square_{c_k}$.

clearly true. After the first update of the first iteration, we have $\text{dist}[a] = \|s-a\| = d_G(s,a)$ for all $a \in S_{\square_{c_1}}$, hence the property **(2)** is satisfied. Assume the lemma holds for all $i < k$, and we show it also holds in the k-th iteration.

To see the property **(1)**, let $a \in S$ be a point such that $d_G(s,a) \le d_G(s,c_k)$. Consider the moment when the k-th iteration begins. Assume for a contradiction that $\text{dist}[a] > d_G(s,a)$ at that time. Suppose $\pi_G(s,a) = \langle z_0, z_1, \ldots, z_t\rangle$ where $z_0 = s$ and $z_t = a$. Define j as the largest index such that $\text{dist}[z_j] = d_G(s,z_j)$. Note that $j \in \{0,\ldots,t-1\}$ because $\text{dist}[s] = d_G(s,s)$ and $\text{dist}[a] \ne d_G(s,a)$. Therefore, $\text{dist}[z_j] = d_G(s,z_j) < d_G(s,a) \le d_G(s,c_k) \le \text{dist}[c_k]$. This implies $z_j \notin A$ (otherwise it contradicts the fact that c_k is the point in A with the smallest dist-value). It follows $z_j \in S_{\square_{c_i}}$ for some $i < k$, as it got removed from A in some previous iteration. Then by our induction hypothesis and the property **(3)**, we have $\text{dist}[z_{j+1}] = d_G(s,z_{j+1})$ at the end of the i-th iteration and thus at the beginning of the k-th iteration, because $\lambda_{z_{j+1}} = z_j$. However, this contradicts the fact that $\text{dist}[z_{j+1}] > d_G(s,z_{j+1})$. As such, $\text{dist}[a] = d_G(s,a)$ when the k-th iteration begins.

Next, we prove the property **(2)**. For convenience, in what follows, we use A to denote the set A during the k-th iteration (before line 9). We have $S_{\square_{c_k}} = A_{\square_{c_k}}$, since $A = S \backslash (\bigcup_{i=1}^{k-1} S_{\square_{c_i}})$ and $c_k \notin \square_{c_i}$ for all $i < k$ by Fact 2. Let $a \in A_{\square_{c_k}}$ be a point and $r = \lambda_a$. We want to show that $\text{dist}[a] = d_G(s,a)$ after the first update of the k-th iteration. If $r \notin A$, then r got removed from A in the i-th iteration for some $i < k$, namely, $r \in S_{\square_{c_i}}$. By our induction hypothesis and the property **(3)**, we have $\text{dist}[a] = d_G(s,a)$ at the end of i-th iteration (and thus in all the next iterations). So assume $r \in A$ (this implies that $r \ne s$ and thus λ_r exists). In this case, a key observation is that before the first update of the k-th iteration, $\text{dist}[r] = d_G(s,r)$. To see this, let $r' = \lambda_r$ (e.g., see Fig. 1). Note that $\|r'-a\| > 1$, otherwise the path $\pi_G(s,r') \circ \langle r',a\rangle$ would be shorter than $\pi_G(s,r') \circ \langle r',r,a\rangle = \pi_G(s,a)$, contradicting the fact that $\pi_G(s,a)$ is the shortest path from s to a. It follows that

$$d_G(s,a) = d_G(s,r') + d_G(r',a) \ge d_G(s,r') + \|r'-a\| > d_G(s,r') + 1.$$

On the other hand, since $a \in \square_{c_k}$, we have

$$d_G(s,a) \le d_G(s,c_k) + d_G(c_k,a) = d_G(s,c_k) + \|c_k - a\| \le d_G(s,c_k) + 1.$$

Therefore, $d_G(s,r') < d_G(s,c_k)$, and by the property **(1)** we have $\text{dist}[r'] = d_G(s,r')$ when the k-th iteration begins. This further implies $r' \notin A$, since $\text{dist}[r'] = d_G(s,r') < d_G(s,c_k) = \text{dist}[c_k]$ when the k-th iteration begins. Hence, r' got removed from A in the i-th iteration for some $i < k$. Using our induction hypothesis and the property **(3)**, we have $\text{dist}[r] = d_G(s,r)$ at the end of the i-th iteration (and thus in all the next iterations). Note that $r \in \boxplus_{c_k}$, because $r \in \odot_a$. We further have $r \in A_{\boxplus_{c_k}}$, as we assumed $r \in A$. Hence, the first update of the k-th iteration makes $\text{dist}[a] = d_G(s,a)$. This proves the property **(2)**. ◀

Lemma 3 implies that $\mathrm{dist}[a] = d_G(s,a)$ for all $a \in S$ at the end of Algorithm 1. Indeed, any point $a \in S$ belongs to $S_{\square_{c_i}}$ for some $i \in \{1,\dots,m\}$, thus the property **(2)** of Lemma 3 guarantees $\mathrm{dist}[a] = d_G(s,a)$. Next, we check the correctness of the $\mathrm{pred}[\cdot]$ table. We want $\mathrm{dist}[a] = \mathrm{dist}[\mathrm{pred}[a]] + \|\mathrm{pred}[a] - a\|$ for all $a \in S$. However, as mentioned before, the sub-routine Update in general may result in data inconsistency, making this equation false. The next lemma shows this can not happen in our algorithm.

▶ **Lemma 4.** *At any moment of Algorithm 1, we always have* $\mathrm{dist}[a] = \mathrm{dist}[\mathrm{pred}[a]] + \|\mathrm{pred}[a] - a\|$ *for all* $a \in S$.

Now we see that Algorithm 1 correctly computes shortest paths from s. However, it is still not clear why simultaneously processing all points in one cell in each iteration makes our algorithm faster than the standard Dijkstra's algorithm. In what follows, we focus on the time complexity of the algorithm. At this point, let us ignore the two Update sub-routines and show how to efficiently implement the remaining part of the algorithm. In each iteration, all the work can be done in constant time except lines 6 and 9. To efficiently implement lines 6 and 9, we maintain the set A in a (balanced) binary search tree using the dist-values as keys. In this way, line 6 can be done in $O(\log n)$ time, and lines 9 can be done in $O(|S_{\square_c}| \cdot \log n)$ time. Note that whenever the dist-value of a point in A is updated, we also need to update the binary search tree in $O(\log n)$ time. This occurs in the two Update sub-routines, which has at most $O(|S_{\square_c}| + |S_{\boxplus_c}|) = O(|S_{\boxplus_c}|)$ modifications of the dist-values. Therefore, the time for updating the binary search tree is $O(|S_{\boxplus_c}| \cdot \log n)$. To summarize, the time cost of the i-th iteration, without the Update sub-routines, is $O(|S_{\boxplus_{c_i}}| \cdot \log n)$. Since $\sum_{i=1}^{m} |S_{\boxplus_{c_i}}| \leq 25n$ by Fact 2, the overall time is $O(n \log n)$. In the following two sections, we shall consider the time complexities of the two Update sub-routines. To efficiently implement the first Update is relatively easy, while the second one is more challenging.

2.1 First update

In this section, we show how to implement the first update (line 7) in $O(|S_{\boxplus_c}| \cdot \log n)$ time. As mentioned before, we can obtain the points in $S_{\boxplus_c}$ using the pointer associated to c, and then further find the points in $A_{\boxplus_c}$ and $A_{\square_c}$. After this, we do $\mathrm{dist}'[a] \leftarrow \mathrm{dist}[a]$ for all $a \in A_{\boxplus_c}$. To implement $\textsc{Update}(A_{\boxplus_c}, A_{\square_c})$, the critical step is to find, for every $r \in A_{\square_c}$, a point $p \in A_{\boxplus_c} \cap \odot_r$ that minimizes $\mathrm{dist}'[p] + \|p - r\|$. This is equivalent to searching the weighted nearest-neighbor of r in the unit disk $\odot_r$ (if we regard $A_{\boxplus_c}$ as a weighted dataset where the weight of each point equals its dist′-value). Unfortunately, it is currently not known how to efficiently solve this problem. Therefore, we need to exploit some special property of the problem in hand. An observation here is that c is the point in $A_{\boxplus_c}$ with the smallest dist′-value and all the points in $A_{\square_c}$ are of distance at most 1 to c (because $c \in A_{\square_c}$). Using this observation, we prove the following key lemma.

▶ **Lemma 5.** *Before the first update of each iteration, for all* $r \in A_{\square_c}$, *we have*

$$\arg\min_{a \in A_{\boxplus_c} \cap \odot_r} \{\mathrm{dist}'[a] + \|a - r\|\} = \arg\min_{a \in A_{\boxplus_c}} \{\mathrm{dist}'[a] + \|a - r\|\}.$$

Proof. Let $p = \arg\min_{a \in A_{\boxplus_c} \cap \odot_r} \{\mathrm{dist}'[a] + \|a - r\|\}$. Define $B = A_{\boxplus_c} \backslash (A_{\boxplus_c} \cap \odot_r)$. It suffices to show that $\mathrm{dist}'[p] + \|p - r\| < \mathrm{dist}'[b] + \|b - r\|$ for all $b \in B$. Fix a point $b \in B$ (e.g., see Fig. 2). We have $\|b - r\| > 1$ by construction. On the other hand, since $r \in A_{\square_c}$, we have $c \in \odot_r$ and hence $\|c - r\| \leq 1$. Furthermore, $\mathrm{dist}'[c] \leq \mathrm{dist}'[b]$, because $b \in A$ and c is the point in A with the smallest dist-value (as well as the smallest dist′-value). It follows that

$$\mathrm{dist}'[p] + \|p - r\| \leq \mathrm{dist}'[c] + \|c - r\| < \mathrm{dist}'[b] + \|b - r\|,$$

where the first "$\leq$" follows from the definition of p and the fact that $c \in A_{\square_c} \cap \odot_r$. ◀

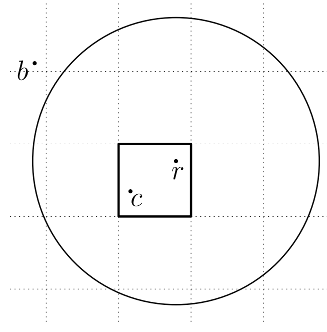


Figure 2 Illustrating the proof of Lemma 5. The solid square is $\square_c$ and the solid circle is $\odot_r$.

The above lemma makes the problem easy. Indeed, for every $r \in A_{\square_c}$, we only need to find a point $p \in A_{\boxplus_c}$ that minimizes $\text{dist}'[p] + \|p - r\|$ and Lemma 5 guarantees that $p \in \odot_r$. This is just the standard (additively-)weighted nearest-neighbor search, which can be solved by building a weighted Voronoi Diagram (WVD) on $A_{\boxplus_c}$ and then querying for each $r \in A_{\square_c}$. Building the WVD takes $O(|A_{\boxplus_c}| \cdot \log |A_{\boxplus_c}|)$ time and linear space [8], and each query can be answered in $O(\log |A_{\boxplus_c}|)$ time. The last step, updating the dist-values and pred-values of the points in $A_{\square_c}$, is easy. So the first update of the i-th iteration can be done in $O(|S_{\boxplus_{c_i}}| \cdot \log n)$ time. Since $\sum_{i=1}^{m} |S_{\boxplus_{c_i}}| \leq 25n$, the total time for the first update is $O(n \log n)$.

2.2 Second update

In this section, we consider the second update (line 8) in Algorithm 1. Unfortunately, the trick used in the first update does not apply, which makes the second update more difficult. Here we design a more general algorithm, which can implement $\text{UPDATE}(U, V)$ for arbitrary subsets $U, V \subseteq S$ in $O(f(k) + k \log k)$ time where $k = |U| + |V|$ and $f(k)$ is the time cost of the OIWNN problem with k operations (i.e., insertions and queries). The framework of the algorithm is presented in Algorithm 2. After copying $\text{dist}[\cdot]$ to $\text{dist}'[\cdot]$, we first sort the points in U in increasing order of their dist'-values (line 2). Then we compute $|U|$ disjoint subsets $V_1, \ldots, V_{|U|}$ of V (line 4), where V_i consists of the points contained in $\odot_{u_i}$ but not contained in $\odot_{u_j}$ for any $j < i$. Note that $\bigcup_{i=1}^{|U|} V_i$ consists of all the points in V who have neighbors in U, and hence we only need to update the shortest-path information of these points. For each point $v \in V_i$, what we do is to find its weighted nearest-neighbor p in $\{u_i, \ldots, u_{|U|}\}$ where the weights are the dist-values (line 9), and update the shortest-path information of v by attempting to use p as predecessor (line 10-12).

We first prove the correctness of Algorithm 2. Consider a point $v \in V_i$. The purpose of $\text{UPDATE}(U, V)$ is to find the weighted nearest-neighbor of v in $U \cap \odot_v$ (and use it to update the shortest-path information of v), while what we find in line 9 is the weighted nearest-neighbor p in $\{u_i, \ldots, u_{|U|}\}$. We notice that $U \cap \odot_v \subseteq \{u_i, \ldots, u_{|U|}\}$ because $v \notin \odot_{u_j}$ for all $j < i$ by the definition of V_i. Therefore, we only need to show that the point p computed by line 9 is contained in $U \cap \odot_v$.

▶ **Lemma 6.** *At line 9 of Algorithm 2, we have $p \in U \cap \odot_v$.*

Proof. Clearly, we have $p \in U$ since $B = \{u_i, \ldots, u_{|U|}\} \subseteq U$. It suffices to show $\|p - v\| \leq 1$. Assume for a contradiction that $\|p - v\| > 1$. We have $\|u_i - v\| \leq 1$ since $v \in V_i$. Furthermore, $\text{dist}'[u_i] \leq \text{dist}'[p]$ because $p \in \{u_i, \ldots, u_{|U|}\}$ and $\text{dist}'[u_i] \leq \text{dist}'[u_j]$ for all $j \geq i$. Hence,

$$\text{dist}'[u_i] + \|u_i - v\| \leq \text{dist}'[u_i] + 1 \leq \text{dist}'[p] + 1 < \text{dist}'[p] + \|p - v\|,$$

which contradicts the fact that p is the weighted nearest-neighbor of v in $\{u_i, \ldots, u_{|U|}\}$. ◀

Algorithm 2 UPDATE(U, V).

1: $\text{dist}'[u] \leftarrow \text{dist}[u]$ for all $u \in U$.
2: Sort the points in $U = \{u_1, \dots, u_{|U|}\}$ such that $\text{dist}'[u_1] \le \dots \le \text{dist}'[u_{|U|}]$
3: **for** $i = 1, \dots, |U|$ **do**
4: $\quad V_i \leftarrow \{v \in V : v \in \odot_{u_i}$ and $v \notin \odot_{u_j}$ for all $j < i\}$
5: $B \leftarrow \emptyset$
6: **for** $i = |U|, \dots, 1$ **do**
7: $\quad B \leftarrow B \cup \{u_i\}$
8: $\quad$**for** $v \in V_i$ **do**
9: $\quad\quad p \leftarrow \arg\min_{b \in B}\{\text{dist}'[b] + \|b - v\|\}$
10: $\quad\quad$**if** $\text{dist}[v] > \text{dist}'[p] + \|p - v\|$ **then**
11: $\quad\quad\quad \text{dist}[v] \leftarrow \text{dist}'[p] + \|p - v\|$
12: $\quad\quad\quad \text{pred}[v] \leftarrow p$

Next, we analyze the time complexity of Algorithm 2. At the beginning, we need to sort the points in U in increasing order of their dist-values, which can be done in $O(|U| \cdot \log |U|)$ time and hence $O(k \log k)$ time. Algorithm 2 basically consists of two loops. We first consider the second loop (line 6-12). In this loop, what we do is weighted nearest-neighbor search on B (line 9) with insertions (line 7), where the weight of each point $b \in B$ is $\text{dist}'[b]$. Note that all insertions and queries here are offline, since the points $u_1, \dots, u_{|U|}$ and the sets $V_1, \dots, V_{|U|}$ are already known before the loop. We have $|U|$ insertions and $|V|$ queries, and hence k operations in total. Recall that $f(k)$ is the time for solving the OIWNN problem with k operations. So this loop takes $f(k)$ time.

Now we consider the first loop (line 3-4). This loop requires us to compute V_i, the subset of V consisting of the points contained in $\odot_{u_i}$ but not contained in $\odot_j$ for all $j < i$, for $i \in \{1, \dots, |U|\}$. We have the following lemma. With the lemma, UPDATE(U, V) can be done in $O(f(k) + k \log k)$ time.

▶ **Lemma 7.** *The first loop of Algorithm 2 takes $O(k \log k)$ time where $k = |U| + |V|$.*

2.3 Putting everything together

As argued before, except the two UPDATE sub-routines, Algorithm 1 runs in $O(n \log n)$ time. Section 2.1 shows that the first update can be done in $O(|S_{\boxplus_c}| \cdot \log n)$ time. Section 2.2 demonstrates that the second update of each iteration can be done in $O(f(k) + k \log k)$ time where $k = |A_{\square_c}| + |A_{\boxplus_c}| = O(|S_{\boxplus_c}|)$ and $f(k)$ is the time for solving the OIWNN problem with k operations. Noting the fact $\sum_{i=1}^m |S_{\boxplus_{c_i}}| \le 25n$, we can conclude the following.

▶ **Theorem 8.** *Suppose the OIWNN problem with k operations can be solved in $f(k)$ time, where $f(k)/k$ is a non-decreasing function. Then there exists an SSSP algorithm in weighted unit-disk graphs with $O(n \log n + f(n))$ running time, where n is the number of the vertices.*

Proof. According to our analysis and the fact $\sum_{i=1}^m |S_{\boxplus_{c_i}}| \le 25n$, the overall time of Algorithm 1 is $O(n \log n + \sum_{i=1}^m f(|S_{\boxplus_{c_i}}|))$. Since $f(k)/k$ is non-decreasing, we have

$$\sum_{i=1}^m f(|S_{\boxplus_{c_i}}|) = \sum_{i=1}^m |S_{\boxplus_{c_i}}| \cdot \frac{f(|S_{\boxplus_{c_i}}|)}{|S_{\boxplus_{c_i}}|} \le \sum_{i=1}^m |S_{\boxplus_{c_i}}| \cdot \frac{f(n)}{n} \le 25 f(n).$$

Therefore, Algorithm 1 runs in $O(n \log n + f(n))$ time. ◀

Using the standard logarithmic method [1] (see also [7] with an additional "bulk update" operation), we can solve the OIWNN problem (even the online version) with k operations in $O(k \log^2 k)$ time using linear space, implying $f(k) = O(k \log^2 k)$. To explore the offline nature of our OIWNN problem, we give in the full version [13] an easier solution with the same performance. By plugging in this algorithm, we obtain the following corollary.

▶ **Corollary 9.** *There exists an SSSP algorithm in weighted unit-disk graphs with $O(n \log^2 n)$ time and $O(n)$ space, where n is the number of the vertices.*

3 The approximation algorithm

We now modify our algorithm framework in the last section (Algorithm 1) to obtain a $(1+\varepsilon)$-approximate algorithm for any $\varepsilon > 0$. Again, let (S, s) be the input of the problem where $|S| = n$ and G be the weighted unit-disk graph induced by S. Formally, a $(1+\varepsilon)$-approximate algorithm computes two tables $\text{dist}[\cdot]$ and $\text{pred}[\cdot]$ indexed by the points in S such that $\text{dist}[a] \leq (1+\varepsilon) \cdot d_G(s,a)$ and $\text{dist}[a] = \text{dist}[\text{pred}[a]] + \|\text{pred}[a] - a\|$ for all $a \in S$. Note that the two tables $\text{dist}[\cdot]$ and $\text{pred}[\cdot]$ enclose, for each point $a \in S$, a path from s to a in G that is a $(1+\varepsilon)$-approximation of the shortest path from s to a.

Our algorithm is shown in Algorithm 3, which differs from our exact algorithm (Algorithm 1) as follows. First, in the initialization, we directly compute the dist-values and pred-values of all the neighbors of s in G (line 4-6); note that if a is a neighbor of s then the shortest path from s to a is $\langle s, a \rangle$, because G is a weighted unit-disk graph. Second, the first update in Algorithm 1 is replaced with two update procedures (line 10-11). Finally, the second update in Algorithm 1 is replaced with an approximate update (line 12) in Algorithm 3, which involves a new sub-routine ApproxUpdate defined as follows. If U and V are two *disjoint* subsets of S, ApproxUpdate(U, V) conceptually does the following.

1. For each $v \in V$, pick a point $p_v \in U \cap \odot_v$ such that $\text{dist}[p_v] + \|p_v - v\| \leq \text{dist}[u] + \|u - v\| + \varepsilon/2$ for all $u \in U \cap \odot_v$.
2. For all $v \in V$, if $\text{dist}[v] < \text{dist}[p_v] + \|p_v - v\|$, then $\text{dist}[v] \leftarrow \text{dist}[p_v] + \|p_v - v\|$ and $\text{pred}[v] \leftarrow p_v$.

Unlike the Update sub-routine, ApproxUpdate cannot result in data inconsistency because we require U and V to be disjoint.

The basic idea of Algorithm 3 is similar to that of our exact algorithm. To verify the correctness of the algorithm, we need to introduce some notations. For $a \in S$, let l_a be the number of the edges on the path $\pi_G(s, a)$ and define $\tau_a = d_G(s,a) + (l_a - 1) \cdot (\varepsilon/2)$. Also, as in Section 2, we use λ_a to denote the s-predecessor of a. We first notice the following fact.

▶ **Fact 10.** *For all $a \in S$, $\tau_a \leq (1+\varepsilon) \cdot d_G(s,a)$.*

Proof. Suppose $\pi_G(s,a) = \langle z_0, z_1, \ldots, z_{l_a} \rangle$ where $z_0 = s$ and $z_{l_a} = a$. Note that $\|z_i - z_{i+2}\| > 1$ for all $i \in \{0, \ldots, l_a - 2\}$, for otherwise $\langle z_0, z_1, \ldots, \hat{z}_{i+1}, \ldots, z_{l_a} \rangle$ would be a shorter path from s to a than $\pi_G(s,a)$ (here $\hat{z}_{i+1}$ means z_{i+1} is absent in the sequence). Therefore, $d_G(s,a) \geq (l_a - 1)/2$, and $\tau_a = d_G(s,a) + (l_a - 1) \cdot (\varepsilon/2) \leq (1+\varepsilon) \cdot d_G(s,a)$. ◀

Let m be the number of iterations of the main loop and c_i be the point c picked in the i-th iteration. Note that Fact 2 also holds for Algorithm 3. Further, we have the following observation, which is similar to Lemma 3 in Section 2.

▶ **Lemma 11.** *Algorithm 3 has the following properties.*

(1) *When the i-th iteration begins, $\text{dist}[a] \leq \tau_a$ for all $a \in S$ with $\tau_a \leq \text{dist}[c_i]$.*

(2) *After line 11 of the i-th iteration, $\text{dist}[a] \leq \tau_a$ for all $a \in S_{\square_{c_i}}$.*

(3) *When the i-th iteration ends, $\text{dist}[a] \leq \tau_a$ for all $a \in S$ with $\lambda_a \in S_{\square_{c_i}}$.*

Algorithm 3 ApproxSSSP(S, s).

1: $\text{dist}[a] \leftarrow \infty$ for all $a \in S$
2: $\text{pred}[a] \leftarrow \text{NIL}$ for all $a \in S$
3: $\text{dist}[s] \leftarrow 0$
4: **for** $a \in (S \backslash \{s\}) \cap \odot_s$ **do**
5: $\quad\text{dist}[a] \leftarrow \|s - a\|$
6: $\quad\text{pred}[a] \leftarrow s$
7: $A \leftarrow S$
8: **while** $A \neq \emptyset$ **do** ▷ Main loop
9: $\quad c \leftarrow \arg\min_{a \in A}\{\text{dist}[a]\}$
10: $\quad\textsc{Update}(A_{\boxplus_c} \backslash A_{\square_c}, A_{\square_c})$
11: $\quad\textsc{Update}(A_{\square_c}, A_{\square_c})$
12: $\quad\textsc{ApproxUpdate}(A_{\square_c}, A_{\boxplus_c} \backslash A_{\square_c})$ ▷ Approximate update
13: $\quad A \leftarrow A \backslash A_{\square_c}$
14: **return** $\text{dist}[\cdot]$ and $\text{pred}[\cdot]$

By the above lemma, we see that $\text{dist}[a] \leq \tau_a$ for all $a \in S$ at the end of Algorithm 3. Indeed, any point $a \in S$ belongs to $S_{\square_{c_i}}$ for some $i \in \{1, \dots, m\}$, thus the property **(2)** of Lemma 11 guarantees that $\text{dist}[a] \leq \tau_a$. Using Fact 10, we further conclude that $\text{dist}[a] \leq (1 + \varepsilon) \cdot d_G(s, a)$ for all $a \in S$ at the end of Algorithm 3. Next, we need to check the correctness of the $\text{pred}[\cdot]$ table. We want $\text{dist}[a] = \text{dist}[\text{pred}[a]] + \|\text{pred}[a] - a\|$ for all $a \in S$. As mentioned in Section 2, the procedure $\textsc{Update}(U, V)$ may result in data inconsistency, namely $\text{dist}[v] > \text{dist}[\text{pred}[v]] + \|\text{pred}[v] - v\|$ for some $v \in V$, when U and V are not disjoint. In Algorithm 3, the only place where this can happen is line 11 (note that the Update sub-routine in line 10 acts on two disjoint sets). However, the following lemma shows that even line 11 cannot result in data inconsistency.

▶ **Lemma 12.** *After line 11 of each iteration in Algorithm 3, we have* $\text{dist}[a] \leq \text{dist}[b] + \|b - a\|$ *for all* $a, b \in A_{\square_c}$*. In particular, at any moment of Algorithm 3, we always have* $\text{dist}[a] = \text{dist}[\text{pred}[a]] + \|\text{pred}[a] - a\|$ *for all* $a \in S$*.*

The correctness of Algorithm 3 is thus proved. Later, the first statement of Lemma 12 will also be used to obtain an efficient implementation of the approximate update.

Next, we consider the time complexity of Algorithm 3. Using the same argument as in Section 2, we see that the running time of Algorithm 3 without line 10-12 is $O(n \log n)$. Line 10 can be implemented using the same method as in Section 2.1, namely building a WVD on the points in $A_{\boxplus_c} \backslash A_{\square_c}$ and querying for each point in $A_{\square_c}$ (the correctness follows from the argument in Section 2.1). Also, line 11 can be implemented in this way, because the points in $A_{\square_c}$ are pairwise adjacent in G. Therefore, the total running time for line 10 and 11 is $O(n \log n)$. It suffices to analyze the time cost of line 12, the approximate update.

3.1 Approximate update

In order to implement the approximate update (line 12) in Algorithm 3, we (implicitly) build another grid Γ' on the plane, which consists of square cells with side-length $\varepsilon/8$. To avoid confusion, we use $\blacksquare$ to denote a cell in Γ'. For a point $a \in S$, let $\blacksquare_a$ denote the cell in Γ' containing a. For a set P of points in $\mathbb{R}^2$ and a cell $\blacksquare$ in Γ', define $P_{\blacksquare} = P \cap \blacksquare$.

Line 12 of Algorithm 3 is ApproxUpdate($A_{\square_c}, A_{\boxplus_c} \backslash A_{\square_c}$). Let $U = A_{\square_c}$ and $V = A_{\boxplus_c} \backslash A_{\square_c}$. We shall use two special properties of the set U: **(i)** all the points in U are contained in one cell in Γ and **(ii)** $\text{dist}[u] \leq \text{dist}[u'] + \|u' - u\|$ for all $u, u' \in U$ before the procedure ApproxUpdate(U, V), which follows from Lemma 12. Our algorithm for implementing ApproxUpdate(U, V) is shown in Algorithm 4, which is a variant of Algorithm 2. Here we no longer need the $\text{dist}'[\cdot]$ table because U and V are disjoint.

Algorithm 4 ApproxUpdate(U, V).

```
1: Sort the points in U = {u_1, ..., u_|U|} such that dist[u_1] ≤ ··· ≤ dist[u_|U|]
2: for i = 1, ..., |U| do
3:     V_i ← {v ∈ V : v ∈ ⊙_{u_i} and v ∉ ⊙_{u_j} for all j < i}
4: U' ← {u_j : j ≥ k for all k such that u_k ∈ ■_{u_j}}
5: B ← ∅
6: for i = |U|, ..., 1 do
7:     if u_i ∈ U' then B ← B ∪ {u_i}
8:     for v ∈ V_i do
9:         p ← arg min_{b∈B} {dist[b] + ||b − v||}
10:        if p ∉ ⊙_v then p ← u_i
11:        if dist[v] > dist[p] + ||p − v|| then
12:            dist[v] ← dist[p] + ||p − v||
13:            pred[v] ← p
```

Recall the definition of the sub-routine ApproxUpdate in Section 3. To verify the correctness of Algorithm 4, it suffices to show that just before line 11, the point p satisfies that $p \in U \cap \odot_v$ and $\text{dist}[p] + \|p - v\| \leq \text{dist}[r] + \|r - v\| + \varepsilon/2$ for all $r \in U \cap \odot_v$. The condition $p \in \odot_v$ is clearly satisfied, because of line 10 (note that $u_i \in \odot_v$). To verify the latter condition, we establish the following lemma.

▶ **Lemma 13.** *Just before line 11 of Algorithm 4, we have* $\text{dist}[p] + \|p - v\| \leq \text{dist}[r] + \|r - v\| + \varepsilon/2$ *for all* $r \in U \cap \odot_v$.

For the time complexity of Algorithm 4, let $k = |U| + |V|$. The sorting in line 1 takes $O(|U| \cdot \log |U|)$ time. The loop in line 2-3 can be implemented in $O(k \log |U|)$ time using the same method as in Section 2.2. In line 4, we can compute the set U' in $O(|U| \cdot \log(|S_{\boxplus_c}|/\varepsilon))$ time by grouping the points in U that belong to the same Γ'-cell (see the full version [13] for a more detailed discussion). The loop in line 6-13 is basically weighted nearest-neighbor search (line 9) with insertions (line 7). There are $O(|V|)$ queries and $O(|U'|)$ insertions. Note that $|U'| = O(\varepsilon^{-2})$, because of the property **(i)** of U. Therefore, if we use $f(k_1, k_2)$ to denote the time cost for solving the OIWNN problem with k_1 operations in which at most k_2 operations are insertions, then the loop in line 6-13 takes $f(k, O(\varepsilon^{-2}))$ time. In sum, the running time of Algorithm 4 is $O(f(k, O(\varepsilon^{-2})) + k \log k)$ time. Therefore, the approximate update in Algorithm 3 can be done in $O(f(|S_{\boxplus_c}|, O(\varepsilon^{-2})) + |S_{\boxplus_c}| \cdot \log(|S_{\boxplus_c}|/\varepsilon))$ time.

3.2 Putting everything together

Except the approximate update, Algorithm 1 runs in $O(n \log n)$ time. Section 3.1 shows that the approximate update of each iteration can be done in $O(f(k, O(\varepsilon^{-2})) + k \log k + k \log(1/\varepsilon))$ time where $k = |A_{\square_c}| + |A_{\boxplus_c}| = O(|S_{\boxplus_c}|)$ and $f(k_1, k_2)$ is the time for solving the OIWNN problem with k_1 operations in which at most k_2 operations are insertions. Noting the fact $\sum_{i=1}^{m} |S_{\boxplus_{c_i}}| \leq 25n$, we can conclude the following.

▶ **Theorem 14.** *Suppose the OIWNN problem with k_1 operations in which at most k_2 operations are insertions can be solved in $f(k_1, k_2)$ time, and assume $f(k_1, k_2)/k_1$ is a non-decreasing function of k_1 for any fixed k_2. Then there exists a $(1+\varepsilon)$-approximate SSSP algorithm in weighted unit-disk graphs with $O(n \log n + n \log(1/\varepsilon) + f(n, O(\varepsilon^{-2})))$ running time, where n is the number of the vertices.*

We give in the full version [13] a linear-space algorithm with $f(k_1, k_2) = O(k_1 \log^2 k_2)$. By plugging in this algorithm, we can obtain the following corollary.

▶ **Corollary 15.** *For any $\varepsilon > 0$, there exists a $(1+\varepsilon)$-approximate SSSP algorithm in weighted unit-disk graphs with $O(n \log n + n \log^2(1/\varepsilon))$ time and $O(n)$ space, where n is the number of the vertices.*

Our algorithm in the above corollary further improves the preprocessing time of the distance oracles in weighted unit-disk graphs given by Chan and Skrepetos [5] (see the full version [13] for details).

References

1. Jou L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8:244–251, 1979.
2. Sergio Cabello and Miha Jejčič. Shortest paths in intersection graphs of unit disks. *Computational Geometry: Theory and Applications*, 48:360–367, 2015.
3. Timothy M. Chan and Dimitrios 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.
4. Timothy M. Chan and Dimitrios Skrepetos. All-Pairs Shortest Paths in Geometric Intersection Graphs. In *Proceedings of the 15th Algorithms and Data Structures Symposium (WADS)*, pages 253–264, 2017.
5. Timothy M Chan and Dimitrios 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. Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86:165–177, 1990.
7. Mark de Berg, Kevin Buchin, Bart M.P. Jansen, and Gerhard Woeginger. Fine-Grained Complexity Analysis of Two Classic TSP Variants. In *Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 5:1–5:14, 2016.
8. Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.
9. Jie Gao and Li Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. *SIAM Journal on Computing*, 35:151–169, 2005.
10. Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha 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.
11. Tomomi Matsui. Approximation algorithms for maximum independent set problems and fractional coloring problems on unit disk graphs. In *Proceedings of Japanese Conference on Discrete and Computational Geometry (JCDCG)*, pages 194–200, 1998.
12. Liam Roditty and Michael Segal. On bounded leg shortest paths problems. *Algorithmica*, 59:583–600, 2011.
13. Haitao Wang and Jie Xue. Near-Optimal Algorithms for Shortest Paths in Weighted Unit-Disk Graphs. *arXiv preprint*, 2019. `arXiv:1903.05255`.