Computing the Minimum Bottleneck Moving Spanning Tree

Haitao Wang
Department of Computer Science, Utah State University, Logan, UT, USA

Yiming Zhao[1]
Department of Computer Science, Utah State University, Logan, UT, USA

Abstract

Given a set P of n points that are moving in the plane, we consider the problem of computing a spanning tree for these moving points that does not change its combinatorial structure during the point movement. The objective is to minimize the bottleneck weight of the spanning tree (i.e., the largest Euclidean length of all edges) during the whole movement. The problem was solved in $O(n^2)$ time previously [Akitaya, Biniaz, Bose, De Carufel, Maheshwari, Silveira, and Smid, WADS 2021]. In this paper, we present a new algorithm of $O(n^{4/3}\log^3 n)$ time.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases minimum spanning tree, moving points, unit-disk range emptiness query, dynamic data structure

Digital Object Identifier 10.4230/LIPIcs.MFCS.2022.82

Related Version *Full Version*: https://arxiv.org/abs/2206.12500

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

1 Introduction

Given a set P of n points in the plane, let G_P be the complete graph whose vertex set is P such that the weight of each edge connecting two points p and q of P is the Euclidean distance between p and q. The *Euclidean minimum spanning tree (EMST)* of P is the spanning tree of G_P with minimum sum of edge weights. The *Euclidean minimum bottleneck spanning tree (EMBST)* of P is the spanning tree of G_P whose largest edge weight is minimized. It is well known that a Delaunay triangulation of P contains an EMST of P [24] and thus an EMST of P can be computed in $O(n\log n)$ time by constructing a Delaunay triangulation of P first. This is also the case for the bottleneck problem.

In this paper, motivated by visualizations of time-varying spatial data [2], we consider a moving version of the EMBST problem where every point of P is moving during a time interval. Without loss of generality, we assume that the time interval is $[0,1]$. A *moving point* $p \in P$ is a continuous function $p : [0,1] \to \mathbb{R}^2$. Let $p(t)$ denote the location of p at time $t \in [0,1]$. We assume that p moves on a straight line segment with a constant velocity, i.e., $p(t)$ is linear in t and points of $\{p(t)|\ t \in [0,1]\}$ form a straight line segment in the plane (see Fig. 1; different points may have different velocities). A *moving spanning tree* T of P connects all points of P and does not change its connection during the whole time interval (i.e., for any two points $p, q \in P$, the path connecting p and q in T always contains the same set of edges). We use $T(t)$ to denote the tree at the time t. The *instantaneous bottleneck* $b_T(t)$ at time t is the maximum length of all edges in $T(t)$. The *bottleneck* $b(T)$ of

[1] Corresponding author.

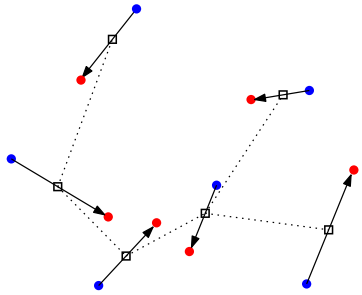

Figure 1 Each pair of red and blue points connected by an arrow represents a moving point. Blue points denote locations at $t = 0$ and red points are locations at $t = 1$. Black boxes are locations of these moving points at certain time and the dashed segments form a spanning tree.

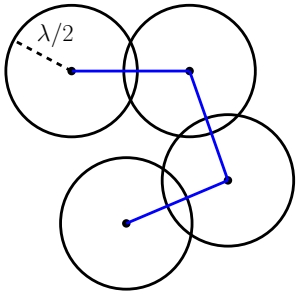

Figure 2 Illustrating a unit-disk graph. Two points are connected (by a blue segment) if their distance is less than or equal to λ. In other words, two points are connected if congruent disks centered at them with radius $\lambda/2$ intersect.

the moving spanning tree T is defined to be the maximum instantaneous bottleneck during the whole time interval, i.e., $b(T) = \max_{t\in[0,1]} b_T(t)$. The *Euclidean minimum bottleneck moving spanning tree* (or *moving-EMBST* for short) T^* refers to the moving spanning tree of P with minimum bottleneck.

In this paper, we study the problem of computing the moving-EMBST T^* for a set P of n moving points in the plane as defined above. Previously, this problem was solved in $O(n^2)$ time by Akitaya, Biniaz, Bose, De Carufel, Maheshwari, Silveira, and Smid [2]. To solve the problem, the authors of [2] first proved the following *key property*: The function of the distance between two moving points over time is convex (this is because each point moves linearly with constant velocity), implying that the maximum distance between two moving points is achieved at $t = 0$ or $t = 1$ (note that this does not mean T^* is attained at either $t = 0$ or $t = 1$; a counterexample is provided in [2]). Using the above property, the authors of [2] proposed the following simple algorithm to compute T^*. First, compute a complete graph G with P as the vertex set such that the weight of each edge connecting two points p and q of P is defined as the maximum length of their distances at $t = 0$ and at $t = 1$. Then the authors [2] showed that a minimum bottleneck spanning tree (MBST) of G is also a moving-EMBST of P and thus it suffices to compute an MBST in G. Since an MBST of a graph can be computed in linear time in the graph size [7], the entire algorithm for computing T^* runs in $O(n^2)$ time in total [2].

1.1 Our result

We present an algorithm of $O(n^{4/3}\log^3 n)$ time to compute T^*. We sketch the main idea below.

For any two points p and q in the plane, let $|pq|$ denote their Euclidean distance. Due to the above key property from [2], we observe that $b(T^*)$ must be equal to $|pq|_{\max}$ for two moving points p and q of P, where $|pq|_{\max} = \max\{|p(0)q(0)|, |p(1)q(1)|\}$, i.e., $b(T^*) \in \{|pq|_{\max} \mid p, q \in P\}$. As such, our main idea is to find $b(T^*)$ in $\{|pq|_{\max} \mid p, q \in P\}$ by binary search. To this end, we first solve a *decision problem*: Given any value $\lambda > 0$, decide whether $b(T^*) \le \lambda$. We reduce the decision problem to the problem of finding a common spanning tree in two unit-disk graphs. Specifically, the *unit-disk graph* $G_\lambda(Q)$ for a set Q of points in the plane with respect to a parameter λ is an undirected graph whose vertex set is Q such that an edge connects two points $p, q \in Q$ if $|pq| \le \lambda$ (alternatively, $G_\lambda(Q)$ can be viewed as the

intersection graph of the set of congruous disks centered at the points of Q with radius $\lambda/2$, i.e., two vertices are connected if their disks intersect; see Fig. 2). Observe that $b(T^*) \leq \lambda$ if and only if the unit-disk graph $G_\lambda(P)$ for P at time $t = 0$ and the unit-disk graph $G_\lambda(P)$ for P at time $t = 1$ share a common spanning tree. To determine whether the two unit-disk graphs share a common spanning tree, we apply breadth-first-search (BFS) on the two graphs simultaneously. To avoid quadratic time, we do not compute these unit-disk graphs explicitly. Instead, we use a batched range searching technique of Katz and Sharir [19] to obtain a compact representation for searching one graph. For searching the other graph, we derive a semi-dynamic data structure for the following *deletion-only unit-disk range emptiness query* problem: Preprocess a set Q of n points in the plane with respect to λ so that the following two operations can be performed efficiently: (1) given a query point p, determine whether Q has a point q such that $|pq| \leq \lambda$, and if yes, return such a point q; (2) delete a point from Q. We refer to the first operation as *unit-disk range emptiness query* (or UDRE query for short). We build a data structure of $O(n)$ space in $O(n \log n)$ time such that each UDRE query can be answered in $O(\log n)$ time while each deletion can be performed in $O(\log n)$ amortized time. This result might be interesting in its own right. Combining this result with the batched range searching [19], we implement the BFS simultaneously on the two unit-disk graphs in $O(n^{4/3} \log^2 n)$ time, which solves the decision problem.

Next, equipped with the above decision algorithm, we find $b(T^*)$ from the set $\{|pq|_{\max} \mid p, q \in P\}$ by binary search. Computing the set explicitly would take $\Omega(n^2)$ time. We avoid doing so by resorting to the distance selection algorithm of Katz and Sharir [19], which can compute the k-th smallest distance among all interpoint distances of a set of n points in the plane in $O(n^{4/3} \log^2 n)$ time for any k with $1 \leq k \leq \binom{n}{2}$. Combining with our decision algorithm, $b(T^*)$ can be computed in $O(n^{4/3} \log^3 n)$ time. Applying the value $\lambda = b(T^*)$ to the decision algorithm can produce the optimal spanning tree T^* in additional $O(n^{4/3} \log^2 n)$ time.

1.2 Related work

Similar to the moving-EMBST problem, one can consider the Euclidean minimum moving spanning tree (moving-EMST) for a set of moving points (i.e., minimizing the total sum of the edge weights instead). The authors of [2] proved that the moving-EMST problem is NP-hard and they gave an $O(n^2)$ time 2-approximation algorithm and another $O(n \log n)$ time $(2+\epsilon)$-approximation algorithm for any $\epsilon > 0$. These spanning tree problems for moving points are relevant in the realm of moving networks that is motivated by the increase in mobile data consumption and the network architecture containing mobile nodes [2].

Geometric problems for moving objects have been studied extensively in the literature, e.g., [3, 4]. In particular, kinetic data structures were proposed to maintain the minimum spanning tree for moving points in the plane [3, 25]. Different from our problem, research in this domain focuses on bounds of the number of combinatorial changes in the minimum spanning tree during the point movement [4].

For solving the deletion-only UDRE query problem, by the standard lifting transformation, one can reduce the problem to maintaining the lower envelope of a dynamic set of planes in $\mathbb{R}^3$, which has been extensively studied [1, 9, 15, 18]. Applying Chan's recent work [11] for the problem can achieve the following result: With $O(n \log n)$ preprocessing time, each UDRE query can be answered in $O(\log^2 n)$ time and each point deletion can be handled in $O(\log^4 n)$ amortized time (the data structure is actually fully-dynamic and can also handle each point insertion in $O(\log^2 n)$ amortized time). The same problem in 2D (whose dual problem becomes maintaining the convex hull for a dynamic set of points) is easier and

has also been studied extensively, e.g., [5, 8, 17, 23]. In addition, Wang [26] studied the unit-disk range counting query problem for a static set of points in the plane, by extending the techniques for half-plane range counting query problem [10, 20, 21].

Our algorithm for the decision problem uses some techniques for unit-disk graphs. Many problems on unit-disk graphs have been studied, i.e., shortest paths and reverse shortest paths [6, 12, 13, 27–30], clique [14], independent set [22], diameter [12, 13, 16], etc. Although a unit-disk graph of n vertices may have $\Omega(n^2)$ edges, many problems can be solved in subquadratic time by exploiting its underlying geometric structures, e.g., computing shortest paths [6, 27]. Our $O(n^{4/3}\log^2 n)$ time algorithm for finding a common spanning tree in two unit-disk graphs adds one more problem to this category.

Outline. In the following, we present our algorithm for the moving-EMBST problem in Section 2. The algorithm uses our data structure for the deletion-only unit-disk range emptiness query problem, which is given in Section 3. Section 4 concludes. Due to the space limit, some proofs are omitted but can be found in the full paper.

2 Algorithm for moving-EMBST

We follow the notation in Section 1, e.g., P, t, $b(T)$, $b_T(t)$, T^*, $|pq|$, $|pq|_{\max}$, $G_\lambda(P)$, etc. Given a set P of n points in the plane, our goal is to compute $b(T^*)$. As discussed in Section 1.1, we first consider the decision problem: Given any $\lambda > 0$, decide whether $b(T^*) \leq \lambda$. We refer to the original problem for computing $b(T^*)$ as the *optimization problem*. In what follows, we solve the decision problem in Section 2.1 and the algorithm for the optimization problem is described in Section 2.2.

2.1 The decision problem

Given any $\lambda > 0$, the decision problem is to decide whether $b(T^*) \leq \lambda$.

For any time $t \in [0, 1]$, we use $P(t)$ to denote the set of points of P at their locations at time t, i.e., $P(t) = \{p(t) \mid p \in P\}$. Consider the two unit-disk graphs $G_\lambda(P(0))$ and $G_\lambda(P(1))$. To simplify the notation, we use $G_\lambda(t)$ to refer to $G_\lambda(P(t))$ for any $t \in [0, 1]$. For every point $p \in P$, we consider $p(0)$ in $G_\lambda(0)$ and $p(1)$ in $G_\lambda(1)$ as the same vertex p, and thus define $G_\lambda = G_\lambda(0) \cap G_\lambda(1)$ as the *intersection graph* of $G_\lambda(0)$ and $G_\lambda(1)$, i.e., the vertex set of G_λ is P and G_λ has an edge connecting two vertices p and q if and only $G_\lambda(0)$ has an edge connecting $p(0)$ and $q(0)$ and $G_\lambda(1)$ has an edge connecting $p(1)$ and $q(1)$. A spanning tree of G_λ is called a *common spanning tree* of $G_\lambda(0)$ and $G_\lambda(1)$.

The following observation has been proved in [2].

▶ **Observation 1** ([2]). $\max_{t\in[0,1]} |p(t)q(t)| = \max\{|p(0)q(0)|, |p(1)q(1)|\}$ *holds for every pair of points* $p, q \in P$.

Using the above observation, the following lemma reduces the decision problem to the problem of finding a common spanning tree of $G_\lambda(0)$ and $G_\lambda(1)$. The proof can be found in the full paper.

▶ **Lemma 2.** *Given any* $\lambda > 0$, $b(T^*) \leq \lambda$ *if and only if* $G_\lambda(0)$ *and* $G_\lambda(1)$ *have a common spanning tree.*

In light of Lemma 2, to solve the decision problem, it suffices to determine whether $G_\lambda(0)$ and $G_\lambda(1)$ have a common spanning tree, or alternatively, whether the intersection graph G_λ has a spanning tree, which is true if and only if the graph is connected. To determine

whether G_λ is connected, we perform a breadth-first search (BFS) in G_λ, or equivalently, we perform a BFS on $G_\lambda(0)$ and $G_\lambda(1)$ simultaneously; we do so without computing the two unit-disk graphs explicitly to avoid the quadratic time. Our algorithm relies on the following lemma for the deletion-only UDRE query problem, which will be proved in Section 3.

▶ **Theorem 3.** *Given a value λ and a set Q of n points in the plane, we can build a data structure of $O(n)$ space in $O(n \log n)$ time such that the following first operation can be performed in $O(\log n)$ worst case time while the second operation can be performed in $O(\log n)$ amortized time.*

1. Unit-disk range emptiness (UDRE) query*: Given a point p, determine whether there exists a point $q \in Q$ such that $|pq| \leq \lambda$, and if yes, return such a point q.*
2. Deletion*: delete a point from Q.*

In the following, we begin with an algorithm overview and then flesh out the details.

Algorithm overview. Starting from an arbitrary point $s \in P$, we run BFS in the graph G_λ. For each $i = 0, 1, 2, \ldots$, let P_i be the set of points whose shortest path lengths from s in G_λ are equal to i. In each i-th iteration, the algorithm computes P_i. Initially, $P_0 = \{s\}$. The algorithm stops once we have $P_i = \emptyset$, after which we check whether all points of P have been discovered. If yes, then the BFS tree is a spanning tree of G_λ; otherwise, G_λ is not connected. Consider the i-th iteration. Suppose P_{i-1} is already known. For each point $p \in P_{i-1}$, we wish to find the set $S(p)$ of all points $q \in P$ such that (1) q has not been discovered yet, i.e., $q \notin \bigcup_{j=0}^{i-1} P_j$; (2) $|p(0)q(0)| \leq \lambda$; (3) $|p(1)q(1)| \leq \lambda$. To implement this step efficiently, we use two techniques. First, we use a batched range searching technique of Katz and Sharir [19] to obtain a compact representation of all points of $P(0)$. The compact representation can provide us with a collection $\mathcal{N}(p)$ of canonical subsets of P whose union is exactly the subset of points q of P such that $|p(0)q(0)| \leq \lambda$. Second, for each subset Q of $\mathcal{N}(p)$, a data structure of Theorem 3 is constructed for $Q(1) = \{q(1) \mid q \in Q\}$, i.e., the set of points of Q at their locations at time $t = 1$. Then, we apply the UDRE query with $p(1)$ as the query point; if the query returns a point $q(1)$, then we know that q is in $S(p)$ and we delete q from Q (we also delete q from other canonical subsets of the compact representation that contain q; the deletion guarantees that points of P already discovered by the BFS have been removed from the canonical subsets of the compact representation) and applying the UDRE query with $p(1)$ again. We keep doing this until the UDRE query does not return any point, and then we process the next subset of $\mathcal{N}(p)$ in the same way. In this way, $S(p)$ will be computed, which is a subset of P_i. Processing every point $p \in P_{i-1}$ as above will produce P_i. The details of the algorithm are given below.

Preprocessing. Before running BFS, we conduct some preprocessing work.

First, using a batched range searching technique [19], we have the following lemma (which is essentially Theorem 3.3 in [19]) for computing a *compact representation* of all pairs (p, q) of points of P with $|p(0)q(0)| \leq \lambda$.

▶ **Lemma 4** (Theorem 3.3 [19])**.** *We can compute a collection $\{X_r \times Y_r\}_r$ of complete edge-disjoint bipartite graphs in $O(n^{4/3} \log n)$ time and space, where $X_r, Y_r \subseteq P$, with the following properties.*

1. *For any r, $|p(0)q(0)| \leq \lambda$ holds for any point $p \in X_r$ and any point $q \in Y_r$.*
2. *The number of these complete edge-disjoint bipartite graphs is $O(n^{4/3})$, and both $\sum_r |X_r|$ and $\sum_r |Y_r|$ are bounded by $O(n^{4/3} \log n)$.*
3. *For any two points $p, q \in P$ with $|p(0)q(0)| \leq \lambda$, there exists a unique r such that $p \in X_r$ and $q \in Y_r$.*

We refer to each X_r (resp., Y_r) as a *canonical subset* of P. After the collection $\{X_r \times Y_r\}_r$ is computed, we further do the following. For each point $p \in P$, if p is in X_r, then we add (the index of) Y_r to $\mathcal{N}(p)$. By Lemma 4(3), subsets of $\mathcal{N}(p)$ are pairwise disjoint and the union of them is exactly the subset of points $q \in P$ with $|p(0)q(0)| \leq \lambda$. Similarly, for each point $p \in P$, if p is in Y_r, then we add (the index of) Y_r to $\mathcal{M}(p)$. The purpose of having $\mathcal{M}(p)$ is that after a point p is identified in P_i, we will need to remove p from all subsets Y_r that contain p (so $\mathcal{M}(p)$ helps us to keep track of these subsets Y_r). We can compute $\mathcal{N}(p)$ and $\mathcal{M}(p)$ for all points $p \in P$ in $O(n^{4/3} \log n)$ time since both $\sum_r |X_r|$ and $\sum_r |Y_r|$ are $O(n^{4/3} \log n)$ by Lemma 4(2). For the same reason, both $\sum_{p \in P} |\mathcal{N}(p)|$ and $\sum_{p \in P} |\mathcal{M}(p)|$ are bounded by $O(n^{4/3} \log n)$.

In addition, for each canonical subset Y_r, we construct the data structure of Theorem 3 for $Y_r(1) = \{q(1) \mid q \in Y_r\}$, denoted by $\mathcal{D}(Y_r)$. Since $\sum_r |Y_r| = O(n^{4/3} \log n)$, constructing the data structures for all Y_r can be done in $O(n^{4/3} \log^2 n)$ time and $O(n^{4/3} \log n)$ space.

This finishes our preprocessing work, which takes $O(n^{4/3} \log^2 n)$ time in total.

Implementing the BFS algorithm. We next implement the BFS algorithm as overviewed above (we follow the same notation).

For each point $p \in P_{i-1}$, the key step is to compute the subset $S(p)$ of P. We implement this step as follows. For each $Y_r \in \mathcal{N}(p)$, we perform a UDRE query with $p(1)$ on the data structure $\mathcal{D}(Y_r)$. If the query returns a point $q(1)$, then we add q to $S(p)$ and delete $q(1)$ from the data structure $\mathcal{D}(Y'_r)$ for every $Y'_r \in \mathcal{M}(q)$. Next, we perform a UDRE query with $p(1)$ on $\mathcal{D}(Y_r)$ again and repeat the same process as above until the query does not return any point. According to the definitions of $\mathcal{N}(p)$ and $\mathcal{M}(p)$ and also due to the deletions on $\mathcal{D}(Y'_r)$ for all $Y'_r \in \mathcal{M}(q)$, the union of $S(p)$ thus computed for all $p \in P_{i-1}$ is exactly P_i. This finishes the i-th iteration of the BFS algorithm.

For the time analysis, since both $\sum_{p \in P} |\mathcal{N}(p)|$ and $\sum_{p \in P} \mathcal{M}(p)$ are $O(n^{4/3} \log n)$, the total number of UDRE queries and deletions on the data structures $\mathcal{D}(Y_r)$ in the entire algorithm is $O(n^{4/3} \log n)$, which together take $O(n^{4/3} \log^2 n)$ time. Therefore, the BFS algorithm runs in $O(n^{4/3} \log^2 n)$ time.

The following theorem summarizes our result for the decision problem.

▶ **Theorem 5.** *Given a value $\lambda > 0$, we can decide whether $b(T^*) \leq \lambda$ in $O(n^{4/3} \log^2 n)$ time, and if yes, a moving spanning tree T of P with $b(T) \leq \lambda$ can be found in $O(n^{4/3} \log^2 n)$ time.*

2.2 The optimization problem

As discussed in Section 1, by Observation 1, $b(T^*)$ is equal to $|p(0)q(0)|$ or $|p(1)q(1)|$ for two moving points $p, q \in P$. As such, we can compute $b(T^*)$ by searching the two sets $S(0)$ and $S(1)$ using our decision algorithm in Theorem 5, where $S(t)$ is defined as $\{|p(t)q(t)| \mid p, q \in P\}$ for any $t \in [0, 1]$. To avoid explicitly computing $S(0)$ and $S(1)$, which would take $\Omega(n^2)$ time, we resort to the distance selection algorithm of Katz and Sharir [19], which can compute the k-th smallest distance among all interpoint distances of a set of n points in the plane in $O(n^{4/3} \log^2 n)$ time for any k with $1 \leq k \leq \binom{n}{2}$. Combining the distance selection algorithm and our decision algorithm, we can compute $b(T^*)$ in $O(n^{4/3} \log^3 n)$ time by doing binary search on the values of $S(0)$ and $S(1)$. The details are given in the proof of the following theorem, which can be found in the full paper.

▶ **Theorem 6.** *Given a set P of n moving points in the plane, we can compute a Euclidean minimum bottleneck moving spanning tree for them in $O(n^{4/3} \log^3 n)$ time.*

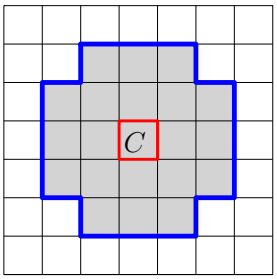

Figure 3 The cells in the gray region bounded by the blue curve are all neighbors of the red cell.

3 Deletion-only unit-disk range emptiness query data structure

In this section, we prove Theorem 3. We follow the notation in the theorem, e.g., Q, λ.

We use a *unit-disk* to refer to a disk with radius λ. For any point p in the plane, we use A_p to denote the unit-disk centered at p. With this notation, a unit-disk range emptiness (UDRE) query with query point p becomes the following: Determine whether $A_p \cap Q$ is empty, and if not, return a point from $A_p \cap Q$.

We use a grid Ψ_λ to capture the neighboring information of the points of Q, which partitions the plane into square cells of side length $\lambda/\sqrt{2}$ by horizontal and vertical lines, so that the distance of any two points in each cell is at most λ. For ease of discussion, we assume that each point of Q is in the interior of a cell of Ψ_λ. Define $Q(C)$ as the subset of points of Q lying in a cell C. A cell C' of Ψ_λ is a *neighbor* of another cell C if the minimum distance between a point of C and a point of C' is at most λ (see Fig. 3). For each cell C, we use $N(C)$ to denote the set of neighbors of C in Ψ_λ; for convenience, we let $N(C)$ include C itself. Note that the number of neighbors of each cell of Ψ_λ is $O(1)$ and each cell is a neighbor of $O(1)$ cells (since $C' \in N(C)$ if and only if $C \in N(C')$). Let $\mathcal{C}$ denote the set of cells of Ψ_λ that contain at least one point of Q as well as their neighbors. Note that $\mathcal{C}$ has $O(n)$ cells. By the definition of $\mathcal{C}$, the following observation is self-evident.

▶ **Observation 7.** *For any point p in the plane, if p is not in any cell of $\mathcal{C}$, then $A_p \cap Q = \emptyset$.*

The grid technique was widely used in algorithms for unit-disk graphs [12, 27, 29, 30]. The following lemma has been proved in [26].

▶ **Lemma 8** ([26]).
1. *The set $\mathcal{C}$, along with the subsets $Q(C)$ and $N(C)$ for all cells $C \in \mathcal{C}$, can be computed in $O(n \log n)$ time and $O(n)$ space.*
2. *With $O(n \log n)$ time and $O(n)$ space preprocessing, given any point p in the plane, we can do the following in $O(\log n)$ time: Determine whether p is in a cell C of $\mathcal{C}$, and if yes, return C and the set $N(C)$.*

Note that we do not compute the entire grid Ψ_λ but only compute the information in Lemma 8. We next prove Theorem 3 using the information computed in Lemma 8.

Consider a UDRE query with a query point p. By Lemma 8(2), we can determine whether p is in a cell $C \in \mathcal{C}$. If not, by Observation 7, we are done with the query. Below we assume that p is in a cell $C \in \mathcal{C}$. In this case, $A_p \cap Q \neq \emptyset$ if and only if $A_p \cap Q(C') \neq \emptyset$ for a cell $C' \in N(C)$. As such, as $|N(C)| = O(1)$, it suffices to check for each cell $C' \in N(C)$, whether $A_p \cap Q(C') = \emptyset$. In this way, we reduce our original problem for Q to $Q(C')$. As such, below we construct a data structure $\mathcal{D}_C(C')$ for $Q(C')$ with respect to C. Note that we also need to handle deletions for $Q(C')$. Depending on whether $C' = C$, there are two cases.

If $C' = C$, then all points of $Q(C')$ are in the disk A_p and thus we can return an arbitrary point of $Q(C')$ as the answer to the UDRE query. To support the deletions on $Q(C')$, we build a balanced binary search tree $T(C')$ for all points of $Q(C')$ sorted by their indices (we can arbitrarily assign indices to points of Q) as our data structure $\mathcal{D}_C(C')$. In this way, deleting a point from $\mathcal{D}_C(C')$ can be done in $O(\log n)$ time. Therefore, in the case where $C' = C$, we can perform each UDRE query and each deletion in $O(\log n)$ time.

In what follows, we assume that $C' \neq C$, which is our main focus. In this case, C' and C are separated by an axis-parallel line. Without loss of generality, we assume that they are separated by a horizontal line ℓ such that C' is above ℓ and C is below ℓ. We further assume that ℓ contains the upper edge of C. The rest of this section is organized as follows. In Section 3.1, we first present some observations which our approach is based on. We describe our preprocessing algorithm for $Q(C')$ in Section 3.2 while handling the UDRE queries and deletions is discussed in Section 3.3. Section 3.4 finally summarizes everything. In the following, we let $m = |Q(C')|$.

3.1 Observations

Our basic idea is to maintain the portion $\mathcal{U}$ inside C of the lower envelope of the unit-disks centered at points of $Q(C')$. Then, $A_p \cap Q(C') \neq \emptyset$ if and only if p is above $\mathcal{U}$. Determining whether p is above $\mathcal{U}$ can be easily done by binary search because $\mathcal{U}$ is x-monotone. To handle deletions, we borrow an idea from Hershberger and Suri [17] for maintaining the convex hull of a semi-dynamic (deletion-only) set of points in the plane. To make our approach work, we first present some observations in this subsection.

Recall that A_q denotes a unit-disk centered at point q. We use ∂A_q to denote the boundary of A_q, which is a unit-circle. Let $\xi_q = \partial A_q \cap C$, i.e., the portion of the circle ∂A_q inside C. Note that it is possible that $\xi_q = \emptyset$, in which case either $A_q \cap C = \emptyset$ or $C \subseteq A_q$. If $A_q \cap C = \emptyset$, then $|pq| > \lambda$ holds for all points $p \in C$ and thus q can be ignored from constructing our data structure $\mathcal{D}_C(C')$. If $C \subseteq A_q$, then $|pq| \leq \lambda$ always holds for all points $p \in C$ and thus we can process all such points q in the same way as the above case $C' = C$. As such, in the following we assume that $\xi_q \neq \emptyset$ for every point $q \in Q(C')$. Because the radius of A_q is λ while the side-length of C is $\lambda/\sqrt{2}$, ξ_q consists of at most two arcs of ∂A_q. Further, ξ_q has exactly two arcs only if ∂A_q intersects the lower edge of C. For simplicity of discussion, we remove the lower edge from C and make C a bottom-unbounded rectangle (i.e., C's upper edge does not change, its two vertical edges extend downwards to the infinity, and its lower edge is removed); so now C has three edges. In this way, $\xi_q = \partial A_q \cap C$ is always a single arc.

Since q is above the horizontal line ℓ, which contains the upper edge of C, ξ_q must be x-monotone. This means the lower envelope $\mathcal{U}$ of $\Xi = \{\xi_q \mid q \in Q(C')\}$ is also x-monotone (see Fig. 4). We will show that $\mathcal{U}$ can be computed in linear time by a Graham's scan style algorithm once the arcs of Ξ are ordered in a certain way. To define this special order, we first introduce some notation below.

Recall that the boundary ∂C consists of three edges. Let l^* denote the lower endpoint of the left edge of C at $-\infty$; similarly, let r^* denote the lower endpoint of the right edge of C (see Fig. 4). For any two points a and b on ∂C, we say that a is *left of* b if a is counterclockwise from b around C (i.e., if we traverse from l^* to r^* along ∂C, a will be encountered earlier than b). For each arc ξ_q, if a and b are its two endpoints and a is left of b (see Fig. 4), then we call a the *left endpoint* of ξ_q and b the *right endpoint*. For ease of exposition, we make a general position assumption that no two arcs of Ξ share a common endpoint. The special order mentioned above for the Graham's scan style algorithm is the order of arcs of Ξ by their right endpoints on ∂C, called *right-endpoint left-to-right order*. To justify the correctness, we prove some properties for the lower envelope $\mathcal{U}$ below.

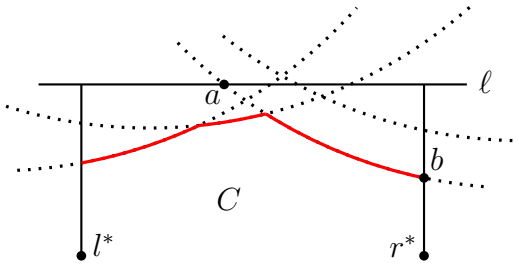

Figure 4 Illustrating the lower envelope (the red curve).

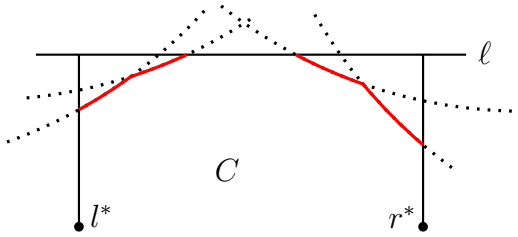

Figure 5 Illustrating a lower envelope (the red curve) that has two connected components.

Suppose we traverse on ∂C from l^* until we meet $\mathcal{U}$, and then we traverse on $\mathcal{U}$ until we come back on ∂C again. We keep traversing. We may meet $\mathcal{U}$ again if $\mathcal{U}$ has multiple connected components (see Fig. 5). We continue in this way until we arrive at r^*. The order of the arcs of Ξ that appear on $\mathcal{U}$ encountered during the above traversal is called the *traversal order* of $\mathcal{U}$. The following is a crucial lemma that our algorithm relies on. The proof can be found in the full paper.

▶ **Lemma 9.** *Every arc of Ξ has at most one portion on $\mathcal{U}$ and the traversal order of $\mathcal{U}$ is consistent with the right-endpoint left-to-right order of Ξ (i.e., if an arc ξ appears in the front of another arc ξ' in the traversal order of $\mathcal{U}$, then the right endpoint of ξ is to the left of that of ξ').*

3.2 Preprocessing

We perform the following preprocessing algorithm for $Q(C')$. Due to Lemma 9, we are able to extend to our problem a technique from Hershberger and Suri [17] for maintaining the convex hull for a semi-dynamic (deletion-only) set of points in the plane (in the dual plane, the problem is to maintain the lower/upper envelope for a semi-dynamic set of lines). Recall that $m = |Q(C')|$.

We first compute the arcs of Ξ and sort them by their right endpoints from left to right on ∂C. Let T be a complete binary tree whose leaves correspond to arcs in the above order. For each node v, let $\Xi(v)$ denote the subset of arcs in the leaves of the subtree of T rooted at v.

For any subset Ξ' of Ξ, let $\mathcal{U}(\Xi')$ denote the lower envelope of the arcs of Ξ'. We use a tree $T(\Xi')$ (which can be considered as a subtree of T) to represent $\mathcal{U}(\Xi')$. Initially, we have the tree $T(\Xi)$, and later $T(\Xi)$ is modified due to point deletions from $Q(C')$ (and correspondingly arc deletions from Ξ). The tree $T(\Xi')$ is defined as follows. For each arc $\xi \in \Xi'$, we copy the leaf of T storing ξ along with all ancestors of the leaf into $T(\Xi')$. If we define $\Xi'(v) = \Xi(v) \cap \Xi'$ for any node v of T, then v is copied into $T(\Xi')$ if and only if $\Xi'(v) \neq \emptyset$. Later we will add some additional node-fields to $T(\Xi')$ to represent the lower envelope $\mathcal{U}(\Xi')$. We call $T(\Xi')$ an *envelope tree*.

We wish to have each node v of $T(\Xi')$ represent the lower envelope $\mathcal{U}(\Xi'(v))$ of arcs of $\Xi'(v)$, i.e., arcs stored in the leaves of the subtree of $T(\Xi')$ rooted at v. We add a node-field $arcs(v)$ for that purpose. Storing the entire lower envelope $\mathcal{U}(\Xi'(v))$ at each $arcs(v)$ of $T(\Xi')$ leads to superlinear total space. To achieve $O(m)$ space, we use the following standard approach (which has been used elsewhere, e.g., [17, 23]): For each arc ξ stored in a leave $v \in T(\Xi')$, ξ is stored only at $arcs(u)$ for the highest ancestor u of v in $T(\Xi')$ such that ξ

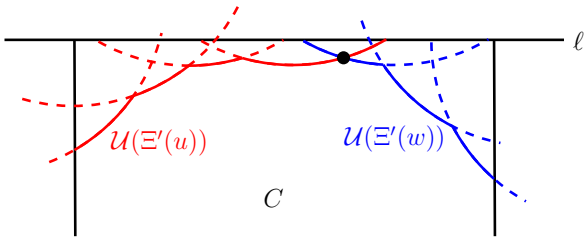

Figure 6 Illustrating Lemma 10: The red (resp., blue) arcs are those from $\Xi'(u)$ (resp., $\Xi'(w)$). There is only one intersection between $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$.

contributes an arc in the lower envelope $\mathcal{U}(\Xi'(u))$. Arcs of $arcs(v)$ in each node v of $T(\Xi')$ are stored in a doubly linked list. Note that if v is the root of $T(\Xi')$, then $arcs(v)$ stores the whole lower envelope $\mathcal{U}(\Xi')$ of Ξ'.

The following lemma, which can be easily obtained from Lemma 9, is crucial to the success of our approach. The proof can be found in the full paper.

▶ **Lemma 10.** *For each node $v \in T(\Xi')$, the lower envelopes $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$ have at most one intersection, where u and w are the left and right children of v, respectively (see Fig. 6).*

By Lemma 10, we add another node-field $X(v)$ for each node $v \in T(\Xi')$ to store the two arcs that define the intersection of $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$, where u and w are the left and right children of v in $T(\Xi')$, respectively. If $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$ do not intersect, then $X(v)$ stores the rightmost arc of $\mathcal{U}(\Xi'(u))$ and the leftmost arc of $\mathcal{U}(\Xi'(w))$. As will be seen later in Section 3.3, the two node-fields $X(v)$ and $arcs(v)$ in $T(\Xi')$ allow us to efficiently maintain the envelope tree $T(\Xi')$ subject to deletions of arcs. We next have the following lemma for constructing $T(\Xi)$ initially.

▶ **Lemma 11.** *Given the set Ξ of m arcs, we can build the envelope tree $T(\Xi)$ in $O(m \log m)$ time.*

Proof. First of all, we can construct the tree T in $O(m \log m)$ time by sorting the arcs of Ξ by their right endpoints on ∂C. The rest of the work is thus to compute the fields $arcs(v)$ and $X(v)$ for all nodes v of T. This can be done in a bottom-up manner as follows.

At the outset, we have $arcs(v) = \Xi(v) = \{\xi\}$ for each leaf node $v \in T$, where ξ is the arc stored at v. We also set $X(v)$ to null. Next, we compute $arcs(\cdot)$ and $X(\cdot)$ for other nodes by merging the lower envelopes of their children. Specifically, consider a node v whose left and right children are u and w, respectively. We assume that $arcs(u)$ and $arcs(w)$ store the lower envelopes $\mathcal{U}(\Xi(u))$ and $\mathcal{U}(\Xi(w))$ in their traversal orders, respectively. The first thing is to compute the lower envelope $\mathcal{U}(\Xi(v))$. By Lemma 10, $\mathcal{U}(\Xi(u))$ and $\mathcal{U}(\Xi(w))$ have at most one intersection. Since each lower envelope is x-monotone, $\mathcal{U}(\Xi(v))$, which is also the lower envelope of $\mathcal{U}(\Xi(u))$ and $\mathcal{U}(\Xi(w))$, can be computed by a standard line sweep procedure. Specifically, a vertical sweeping line ℓ' sweeps the plane from left to right. During the sweeping, we maintain the two arcs of $\mathcal{U}(\Xi(u))$ and $\mathcal{U}(\Xi(w))$ intersecting ℓ', respectively. An event happens if ℓ' hits a vertex of either $\mathcal{U}(\Xi(u))$ or $\mathcal{U}(\Xi(w))$. The sweeping procedure takes $O(|\Xi(v)|)$ time (note that $\Xi(v) = \Xi(u) \cup \Xi(w)$).

- If $\mathcal{U}(\Xi(u))$ and $\mathcal{U}(\Xi(w))$ do not have any intersection, then $\mathcal{U}(\Xi(v))$ is just the concatenation of $\mathcal{U}(\Xi(u))$ and $\mathcal{U}(\Xi(w))$, i.e., we concatenate $arcs(u)$ and $arcs(w)$ and store the result at $arcs(v)$; we also need to reset both $arcs(u)$ and $arcs(w)$ to null. In addition, $X(v)$ is set to including the rightmost arc of $\mathcal{U}(\Xi(u))$ and the leftmost arc of $\mathcal{U}(\Xi(w))$.

- If $\mathcal{U}(\Xi(u))$ and $\mathcal{U}(\Xi(w))$ have an intersection, say, a^*, then let $\xi_u \in \mathcal{U}(\Xi(u))$ and $\xi_v \in \mathcal{U}(\Xi(v))$ be the two arcs that intersect at a^*. We concatenate the part of $\mathcal{U}(\Xi(u))$ left to a^* and the part of $\mathcal{U}(\Xi(w))$ right to a^* (ξ_u and ξ_w are cut off at a^*); the result is $\mathcal{U}(\Xi(v))$ and we store it into $arcs(v)$. Further, arcs left to a^* (including ξ_u) in $\mathcal{U}(\Xi(u))$ and arcs right to a^* (including ξ_w) in $\mathcal{U}(\Xi(w))$ are removed from $arcs(u)$ and $arcs(w)$, respectively. In addition, $X(v)$ is set to $\{\xi_u, \xi_w\}$.

As such, computing the node-fields of v takes $O(|\Xi(v)|)$ time. Doing this for all nodes v in the same level of the tree takes $O(m)$ time as the union of $\Xi(v)$ of all nodes v in the same level is exactly Ξ. Therefore, the construction of the envelope tree $T(\Xi)$ can be done in $O(m \log m)$ time in total. ◀

The above finishes our preprocessing for the points $Q(C')$, which takes $O(m \log m)$ time and $O(m)$ space. Our preprocessing builds the envelope tree $T(\Xi)$, which is our data structure $\mathcal{D}_C(C')$. Once points from $Q(C')$ are deleted we use Ξ' to refer to the subset of Ξ defined by the remaining points and use $T(\Xi')$ to refer to the corresponding envelope tree.

3.3 Handling UDRE queries and point deletions

We now discuss how to handle the UDRE queries and point deletions.

UDRE queries. Handling the UDRE queries is relatively easy. Consider a query point p in the cell C. We wish to determine whether $A_p \cap Q(C') = \emptyset$, and if not, return a point $q \in A_p \cap Q(C')$. Let Ξ' be the set of arcs defined by the points in the current set $Q(C')$. As discussed before, it suffices to determine whether p is above the lower envelope $\mathcal{U}(\Xi')$. To this end, since $\mathcal{U}(\Xi')$ is x-monotone, let a and b be the two adjacent vertices of $\mathcal{U}(\Xi')$ such that p's x-coordinate is between those of a and b. Let ξ_q be the arc that contains the portion of $\mathcal{U}(\Xi')$ between a and b, where q is the center of the arc (and thus $q \in Q(C')$). As such, p is above $\mathcal{U}(\Xi')$ if and only if p is above ξ_q (i.e., p is inside the unit-disk A_q). If yes, then $q \in A_p \cap Q(C')$ and thus we can return q as the answer to the query. Therefore, it suffices to compute the arc ξ_q. To this end, one may attempt to perform binary search on the vertices of $\mathcal{U}(\Xi')$ to find a and b first. However, although the whole $\mathcal{U}(\Xi')$ is stored in $arcs(v)$ at the root v, arcs of $arcs(v)$ are stored in a doubly linked list, which does not support binary search. To circumvent the issue, we can actually perform binary search using the node-fields $X(\cdot)$ of $T(\Xi')$ as follows.

Observe that each vertex of $\mathcal{U}(\Xi')$ appears as the intersection of the two arcs of $X(v)$ for some node $v \in T(\Xi')$. The subtree of $T(\Xi')$ rooted at any node v represents $\mathcal{U}(\Xi'(v))$ by the intersections of the arcs of $X(\cdot)$ stored at its nodes. To find ξ_q, starting from the root, for each node v of $T(\Xi')$, we compute the intersection a^* of the arcs of $X(v)$. If the x-coordinate of p is smaller or equal to that of a^*, we proceed on the left subtree of v recursively; otherwise, we proceed on the right subtree. At the end we will reach a leaf and the arc stored at the leaf is ξ_q. As such, ξ_q can be found in $O(\log m)$ time.

Therefore, each UDRE query can be answered in $O(\log m)$ time.

Deletions. Next, we discuss point deletions. To delete a point q from $Q(C')$, it boils down to deleting the arc ξ_q defined by q from the envelope tree $T(\Xi')$. The next lemma provides an algorithm for this.

▶ **Lemma 12.** *Deleting an arc from the envelope tree $T(\Xi')$ can be done in $O(\log m)$ amortized time.*

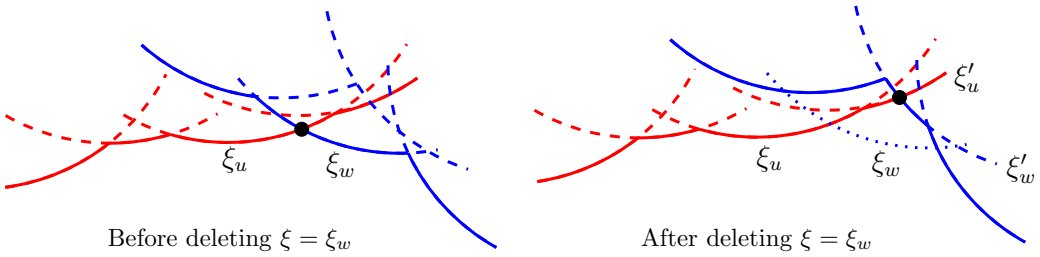

Figure 7 Illustrating the deletion of $\xi = \xi_w$. The red (resp., blue) arcs are those from $\Xi'(u)$ (resp., $\Xi'(w)$).

Proof. Let ξ be the arc we wish to delete from $T(\Xi')$ and let z be the leaf node of the tree storing ξ. To delete ξ, we need to update $arcs(\cdot)$ and $X(\cdot)$ for all ancestors of z.

The algorithm is recursive. Starting from the root, for each node v, we process it by calling Delete(ξ, v) as follows. We assume that $arcs(v)$ now stores the whole lower envelope $\mathcal{U}(\Xi'(v))$, which is true initially when v is the root. Let u and w denote the left and right children of v, respectively. We assume that the leaf z is in the right subtree of v since the other case is symmetric. Let $X(v) = \{\xi_u, \xi_w\}$, with $\xi_u \in \mathcal{U}(\Xi'(u))$ and $\xi_w \in \mathcal{U}(\Xi'(w))$, i.e., the intersection of ξ_u and ξ_w, denoted by a^*, is the intersection between $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$. We first restore $\mathcal{U}(\Xi'(u))$, by concatenating the part of $arcs(v)$ left to a^* and $arcs(u)$. Restoring $\mathcal{U}(\Xi'(w))$ can be done in a similar way. Depending on whether $w = z$, there are two cases.

If w is the leaf z (which is the base case of our recursive algorithm), then $arcs(w) = \{\xi\}$ and we reset the right child of v and field $X(v)$ to null. We also reset $arcs(v) = arcs(u)$ and $arcs(u) = null$.

If w is not z, then to update $arcs(v)$ and $X(v)$, observe that if $\xi \notin X(v)$, then deleting ξ does not affect the intersection between $\mathcal{U}(\Xi'(u))$ and the new lower envelope $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$, i.e., $X(v)$ does not change. Hence, if $\xi \notin X(v)$, we proceed on w by calling Delete(ξ, w). After Delete(ξ, w) is returned, the new $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$ is stored in $arcs(w)$ and we cut $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$ using $X(v)$ to obtain $arcs(v)$ in the same way as the tree construction algorithm in Lemma 11, which takes $O(1)$ time as each $arcs(\cdot)$ is stored by a doubly linked list. In the following, we discuss the case where $\xi \in X(v) = \{\xi_u, \xi_w\}$.

Since ξ is in the right subtree of v, ξ must be ξ_w. In this case, $X(v)$ will be changed after the deletion of ξ and thus we need to compute the new arcs that define the intersection of $\mathcal{U}(\Xi'(u))$ and the new lower envelope $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$ (see Fig. 7). We proceed on w by calling Delete(ξ, w). After Delete(ξ, w) is returned, the new $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$ is stored in $arcs(w)$. Let $\{\xi'_u, \xi'_w\}$ be the new $X(v)$ to be computed, with ξ'_v and ξ'_w in $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$, respectively. Observe that ξ'_u cannot lie to the left of ξ_u in $arcs(u)$ while ξ'_w must lie on the part of the new $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$ between the two old neighbors of ξ $(=\xi_w)$ on $\mathcal{U}(\Xi'(w))$ (see Fig. 7). As such, we compute ξ'_u and ξ'_w using a line sweep procedure that is similar to the algorithm in Lemma 11, but to make the algorithm faster, due to the above observation it suffices to start the sweeping line from the left of the following two arcs: ξ_u and the left neighbor of ξ in the original lower envelope $\mathcal{U}(\Xi'(w))$. We stop the sweeping once the intersection of $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w) \setminus \{\xi\})$ is found, after which, we reset $arcs(v)$ as well as $arcs(u)$ and $arcs(w)$ in constant time in a way similar to the algorithm in Lemma 11.

The pseudocode summarizing the algorithm can be found in the full paper.

For the time analysis, the time we spend on each node v is $O(1)$ except the line sweep procedure for computing ξ'_u and ξ'_w in the case where $\xi \in X(v)$. The procedure takes time $O(1 + k_u + k_w)$, where k_u is the number of arcs between ξ_u and ξ'_u in $\mathcal{U}(\Xi'(u))$ and k_w is

the number of arcs between ξ_w and ξ'_w in $\mathcal{U}(\Xi'(w))$. Observe that the arcs between ξ_u and ξ'_u in $\mathcal{U}(\Xi'(u))$ are moved up from node u to node v after the deletion of ξ (i.e., they were originally stored at $arcs(u)$ but are stored at $arcs(v)$ after the deletion). Similarly, the arcs between ξ_w and ξ'_w in $\mathcal{U}(\Xi'(w))$ are moved up to w from some lower levels after the deletion (see Fig. 7). Because each arc can be moved up at most $O(\log m)$ times for all m point deletions of $Q(C')$, the total sum of $k_u + k_w$ for all deletions is bounded by $O(m \log m)$. As such, each deletion takes $O(\log m)$ amortized time. ◀

3.4 Putting everything together

The above shows that we can build a data structure $\mathcal{D}_C(C')$ for the points of $Q(C')$ with respect to C in $O(m \log m)$ time and $O(m)$ space, such that each UDRE query with a query point in C can be answered in $O(\log m)$ time and deleting a point from $Q(C')$ can be handled in $O(\log m)$ amortized time.

To solve our original problem on Q, i.e., proving Theorem 3, for each cell $C \in \mathcal{C}$, we build data structures $\mathcal{D}_C(C')$ for all cells $C' \in N(C)$. Because $|N(C)| = O(1)$ for every $C \in \mathcal{C}$ and each cell C' is in $N(C)$ for a constant number of cells $C \in \mathcal{C}$, the total space for all these data structures $\mathcal{D}_C(C')$ is $O(n)$ and the total preprocessing time is $O(n \log n)$.

For each UDRE query with a query point p, we first use Lemma 8(2) to determine whether p is in a cell of $\mathcal{C}$. If not, then by Observation 7, $A_p \cap Q = \emptyset$ and thus we are done with the query. Otherwise, Lemma 8(2) will return the cell C that contains p as well as $N(C)$. Then, for each $C' \in N(C)$, we solve the query using the data structure $\mathcal{D}_C(C')$. The total query time is $O(\log n)$ as $|N(C)| = O(1)$.

To delete a point q from Q, using Lemma 8(2) we first find the cell C' that contains q as well as $N(C')$. Notice that $N(C')$ exactly consists of those cells C with $C' \in N(C)$. We then delete q from the data structure $\mathcal{D}_C(C')$ for each $C \in N(C')$. As $|N(C')| = O(1)$, the total deletion time is $O(\log n)$ amortized time. This proves Theorem 3.

4 Conclusion

In this paper, we presented an $O(n^{4/3} \log^3 n)$ time algorithm for computing a Euclidean minimum bottleneck moving spanning tree for a set of n moving points in the plane, which significantly improves the previous $O(n^2)$ time solution [2]. To solve the problem, we first solved the decision problem in $O(n^{4/3} \log^2 n)$ time. This is done by reducing it to the problem of computing a common spanning tree in two unit-disk graphs. To avoid computing the unit-disk graphs explicitly, which would cost $\Omega(n^2)$ time, we used a batched range searching technique [19] to obtain a compact representation for searching one graph, and derived a semi-dynamic (deletion-only) unit-disk range emptiness query data structure for searching the other graph. We believe our data structure is interesting in its own right and will certainly find applications elsewhere. We finally remark that although in our problem each moving point is required to move linearly with constant velocity, our algorithm still works for other types of point movements as long as Observation 1 holds.

References

1 Pankaj K. Agarwal and Jiří Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.

2 Hugo A. Akitaya, Ahmad Biniaz, Prosenjit Bose, Jean-Lou De Carufel, Anil Maheshwari, Luís Fernando Schultz Xavier da Silveira, and Michiel Smid. The minimum moving spanning tree problem. In *Proceedings of the 17th Workshop on Algorithms and Data Structures (WADS)*, pages 15–28, 2021.

3 Mikhail J. Atallah. Dynamic computational geometry. In *Proceedings of 24th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 92–99, 1983.

4 Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999.

5 Gerth S. Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 617–626, 2002.

6 Sergio Cabello and Miha Jejčič. Shortest paths in intersection graphs of unit disks. *Computational Geometry: Theory and Applications*, 48(4):360–367, 2015.

7 Paolo M. Camerini. The min-max spanning tree problem and some extensions. *Information Processing Letters*, 7:10–14, 1978.

8 Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmaic amortized time. *Journal of the ACM*, 48:1–12, 2001.

9 Timothy M. Chan. A dynamic data structure for 3-D convex hulls and 2-D nearest neighbor queries. *Journal of the ACM*, 57:16:1–16:15, 2010.

10 Timothy M. Chan. Optimal partition trees. *Discrete and Computational Geometry*, 47:661–690, 2012.

11 Timothy M. Chan. Dynamic geometric data structures via shallow cuttings. *Discrete and Computational Geometry*, 64:1235–1252, 2020.

12 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.

13 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.

14 Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit disk graphs. *Discrete mathematics*, 86:165–177, 1990.

15 David Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete and Computational Geometry*, 13:111–122, 1995.

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

17 John Hershberger and Subhash Suri. Applications of a semi-dynamic convex hull algorithm. *BIT Numerical Mathematics*, 32(2):249–267, 1992.

18 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.

19 Matthew J. Katz and Micha Sharir. An expander-based approach to geometric optimization. *SIAM Journal on Computing*, 26(5):1384–1408, 1997.

20 Jiří Matoušek. Efficient partition trees. *Discrete and Computational Geometry*, 8:315–334, 1992.

21 Jiří Matoušek. Range searching with efficient hierarchical cuttings. *Discrete and Computational Geometry*, 10:157–182, 1993.

22 Tomomi Matsui. Approximation algorithms for maximum independent set problems and fractional coloring problems on unit disk graphs. In *Proceedings of the Japanese Conference on Discrete and Computational Geometry (JCDCG)*, pages 194–200, 1998.

23 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer System Sciences*, 23(2):166–204, 1981.

24 Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

25 Zahed Rahmati and Alireza Zarei. Kinetic Euclidean minimum spanning tree in the plane. *Journal of Discrete Algorithms*, 16:2–11, 2012.

26 Haitao Wang. Unit-disk range searching and applications. In *Proceedings of the 18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 32:1–32:17, 2022.

27 Haitao Wang and Jie Xue. Near-optimal algorithms for shortest paths in weighted unit-disk graphs. *Discrete and Computational Geometry*, 64:1141–1166, 2020.

28 Haitao Wang and Yiming Zhao. An optimal algorithm for L_1 shortest paths in unit-disk graphs. In *Proceedings of the 33rd Canadian Conference on Computational Geometry (CCCG)*, pages 211–218, 2021.

29 Haitao Wang and Yiming Zhao. Reverse shortest path problem for unit-disk graphs. In *Proceedings of the 17th International Symposium of Algorithms and Data Structures (WADS)*, pages 655–668, 2021.

30 Haitao Wang and Yiming Zhao. Reverse shortest path problem for weighted unit-disk graphs. In *Proceedings of the 16th International Conference and Workshops on Algorithms and Computation (WALCOM)*, pages 135–146, 2022.