5-2023

Algorithms for Unit-Disk Graphs and Related Problems

Yiming Zhao
Utah State University

ALGORITHMS FOR UNIT-DISK GRAPHS AND RELATED PROBLEMS

by

Yiming Zhao

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Computer Science

Approved:

Haitao Wang, Ph.D. Major Professor	David Brown, Ph.D. Committee Member
Curtis Dyreson, Ph.D. Committee Member	Steve Petruzza, Ph.D. Committee Member
Shuhan Yuan, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2023

ABSTRACT

Algorithms for Unit-Disk Graphs and Related Problems

by

Yiming Zhao, Doctor of Philosophy

Utah State University, 2023

Major Professor: Haitao Wang, Ph.D.
Department: Computer Science

In this dissertation, we study algorithms for several problems on unit-disk graphs and related problems. The unit-disk graph can be viewed as an intersection graph of a set of congruent disks. Unit-disk graphs have been extensively studied due to many of their applications, e.g., modeling the topology of wireless sensor networks. Specifically, we consider the following problems: L_1 shortest paths in unit-disk graphs, reverse shortest paths in unit-disk graphs, minimum bottleneck moving spanning trees, unit-disk range reporting, distance selection, etc. We develop efficient algorithms for these problems and our results are either first-known solutions or somehow improve the previous work.

In the problem of L_1 single source shortest path in unit-disk graphs, we are given a point set P and a source point $s \in P$, the target is to find all shortest paths from s to all other vertices in the L_1 weighted unit-disk graph defined on set P. We present an $O(n \log n)$ time algorithm, which matches the $\Omega(n \log n)$-time lower bound. In the second problem, we are given a set P of n points, parameters $r, \lambda > 0$, and two points $s, t \in P$, the goal is to compute the smallest r such that the shortest path length between s and t in the unit-disk graph with respect to set P and parameter r is at most λ. We propose an algorithm of $O(\lfloor \lambda \rfloor \cdot n \log n)$ time and another algorithm of $O(n^{5/4} \log^{7/4} n)$ time for the unweighted case. We also give an $O(n^{5/4} \log^{5/2} n)$ time algorithm for the weighted case. In the third

problem, we are given a set P of n points that are moving in the plane, the problem is to compute a spanning tree for these moving points that does not change its combinatorial structure during the point movement such that the bottleneck weight of the spanning tree (i.e., the largest Euclidean length of all edges) during the whole movement is minimized. We present an algorithm that runs in $O(n^{4/3}\log^3 n)$ time. The fourth problem is unit-disk range reporting in which we are given a set P of n points in the plane and a value r, we need to construct a data structure so that given any query disk of radius r, all points of P in the query disk can be reported efficiently. We build a data structure of $O(n)$ space in $O(n\log n)$ time that can answer each query in $O(k+\log n)$ time, where k is the output size. The time complexity of our algorithm is the same as the previous result but our approach is much simpler. Finally, for the problem of distance selection, we are given a set P of n points in the plane and an integer $1 \leq k \leq \binom{n}{2}$, the target is to find the k-th smallest interpoint distance among all pairs of points of P. We propose an algorithm that runs in $O(n^{4/3}\log n)$ time. Our techniques yield two algorithmic frameworks for solving geometric optimization problems.

Many algorithms and techniques developed in this dissertation are quite general and fundamental, and we believe they will find other applications in future.

(169 pages)

PUBLIC ABSTRACT

Algorithms for Unit-Disk Graphs and Related Problems

Yiming Zhao

In this dissertation, we study algorithms for several problems on unit-disk graphs and related problems. The unit-disk graph can be viewed as an intersection graph of a set of congruent disks. Unit-disk graphs have been extensively studied due to many of their applications, e.g., modeling the topology of wireless sensor networks. Lots of problems on unit-disk graphs have been considered in the literature, such as shortest paths, clique, independent set, distance oracle, diameter, etc. Specifically, we study the following problems in this dissertation: L_1 shortest paths in unit-disk graphs, reverse shortest paths in unit-disk graphs, minimum bottleneck moving spanning tree, unit-disk range reporting, distance selection, etc. We develop efficient algorithms for these problems and our results are either first-known solutions or somehow improve the previous work.

Given a set P of n points in the plane and a parameter $r > 0$, a unit-disk graph $G(P)$ can be defined using P as its vertex set and two points of P are connected by an edge if the distance between these two points is at most r. The weight of an edge is one in the unweighted case and is equal to the distance between the two endpoints in the weighted case. Note that the distance between two points can be measured by different metrics, e.g., L_1 or L_2 metric.

In the first problem of L_1 shortest paths in unit-disk graphs, we are given a point set P and a source point $s \in P$, the problem is to find all shortest paths from s to all other vertices in the L_1 weighted unit-disk graph defined on set P. We present an $O(n \log n)$ time algorithm, which matches the $\Omega(n \log n)$-time lower bound. In the second problem, we are given a set P of n points, parameters $r, \lambda > 0$, and two points s and t of P, the goal is to compute the smallest r such that the shortest path length between s and t in the unit-disk

graph with respect to set P and parameter r is at most λ. This problem can be defined in both unweighted and weighted cases. We propose an algorithm of $O(\lfloor\lambda\rfloor \cdot n \log n)$ time and another algorithm of $O(n^{5/4} \log^{7/4} n)$ time for the unweighted case. We also give an $O(n^{5/4} \log^{5/2} n)$ time algorithm for the weighted case. In the third problem, we are given a set P of n points that are moving in the plane, the problem is to compute a spanning tree for these moving points that does not change its combinatorial structure during the point movement such that the bottleneck weight of the spanning tree (i.e., the largest Euclidean length of all edges) during the whole movement is minimized. We present an algorithm that runs in $O(n^{4/3} \log^3 n)$ time. The fourth problem is unit-disk range reporting in which we are given a set P of n points in the plane and a value r, we need to construct a data structure so that given any query disk of radius r, all points of P in the disk can be reported efficiently. We build a data structure of $O(n)$ space in $O(n \log n)$ time that can answer each query in $O(k + \log n)$ time, where k is the output size. The time complexity of our algorithm is the same as the previous result but our approach is much simpler. Finally, for the problem of distance selection, we are given a set P of n points in the plane and an integer $1 \leq k \leq \binom{n}{2}$, the distance selection problem is to find the k-th smallest interpoint distance among all pairs of points of P. We propose an algorithm that runs in $O(n^{4/3} \log n)$ time. Our techniques yield two algorithmic frameworks for solving geometric optimization problems.

Many algorithms and techniques developed in this dissertation are quite general and fundamental, and we believe they will find other applications in future.

ACKNOWLEDGMENTS

It is with great pleasure, gratitude, and humility that I acknowledge the individuals who have supported and encouraged me throughout my Ph.D. journey in the past four years. Their steadfast support has made this work possible.

First and foremost, I would like to extend my heartfelt thanks to my brilliant supervisor, Dr. Haitao Wang, who has been a constant source of inspiration, support, and encouragement throughout my research journey. Dr. Wang's exceptional guidance, keen insights, and unwavering commitment to academic excellence have been invaluable in shaping my research and scholarship. His dedication, expertise, and vision have challenged me to push beyond my limits, and his mentorship has been instrumental in my academic and personal growth. I am indebted to him for his continuous support.

I would also like to express my gratitude to the other members of my supervisory committee, Dr. David Brown, Dr. Curtis Dyreson, Dr. Steve Petruzza, and Dr. Shuhan Yuan, for their insightful comments and constructive feedback. Their intellectual rigor, scholarly excellence, and steadfast support have been a critical component of my academic journey, and I am deeply grateful for their guidance and suggestions.

Moreover, I extend my sincere thanks to the staff in the Department of Computer Science for their administrative support, without which my research could not have been possible. Their professionalism, efficiency, and dedication to their work have made my academic journey smoother and more enjoyable.

In conclusion, I feel privileged and honored to have the opportunity to work with such exceptional individuals, and I am deeply grateful for their support, encouragement, and guidance.

This work was supported in part by the National Science Foundation through Grant CCF-2005323.

Yiming Zhao

CONTENTS

LIST OF FIGURES

List of Algorithms

CHAPTER 1

INTRODUCTION

We propose and study several problems related to unit-disk graphs, which is an important class of geometric intersection graphs. A great number of problems on unit-disk graphs have been studied in the literature due to many of their applications, e.g., in wireless sensor networks.

In the rest of this chapter, we first briefly introduce the concept of unit-disk graphs and then present an overview of the problems we study in this dissertation. An outline of this dissertation is given at the end of this chapter.

1.1 Unit-disk graphs

A unit-disk graph can be viewed as an intersection graph of a set of congruous disks in the plane, i.e., centers of the disks are vertices and two centers of disks are connected by an edge if two corresponding disks intersect. Unit-disk graphs have been widely studied due to many of their applications, e.g., in wireless sensor networks [2–5]. Lots of problems on unit-disk graphs have been considered in the literature, such as the shortest path problem [1, 6–10], the clique problem [11], the independent set problem [12], distance oracle [7, 8, 13], the diameter problem [7, 8, 13], etc. Comparing to general graphs, these problems in unit-disk graphs can be solved more efficiently by exploiting their underlying geometric structures.

1.2 Overview of our problems

In this section, we give an overview of the problems we study in this dissertation. The details can be found in subsequent chapters.

1.2.1 L_1 shortest paths in unit-disk graphs

This problem comes from the traditional version of the single-source shortest path (SSSP) problem in unit-disk graphs but the distance between two points in the plane is measured by the L_1 metric. The problem can be formulated as follows. Let P be a set of n points in the plane. The L_1 *unit-disk graph* $G(P)$ of P is a graph with P as its vertex set such that two points of P are connected by an edge if the L_1 distance between the two points is at most 1. Alternatively, $G(P)$ is the intersection graph of the set of disks (diamonds in the L_1 case) centered at the points of P with radii equal to $1/2$. Each edge of $G(P)$ has a weight that is equal to the L_1 distance of the two incident vertices of the edge. Given set P and a source point $s \in P$, we compute shortest paths in $G(P)$ from s to all other points of P.

The L_2 case of the SSSP problem where the distance is measured under the L_2 metric has been extensively studied [1, 6–10]. The current best algorithm, which was given by Wang and Xue [1], runs in $O(n \log^2 n)$ time. We follow the algorithmic framework of Wang and Xue [1] but give a faster implementation for the L_1 case by deriving a more efficient algorithm for the bottleneck subproblem in the L_1 case. This leads to an overall $O(n \log n)$-time algorithm, which is optimal.

Our results on this problem have been published in a conference [14] and a journal [15]. Refer to Chapter 2 for the details.

1.2.2 Reverse shortest path problem for unit-disk graphs

Given a set P of n points in the plane and a parameter r, the *unit-disk graph* $G_r(P)$ is an undirected graph whose vertex set is P such that an edge connects two points $p, q \in P$ if the distance between p and q is at most r. The weight of each edge of $G_r(P)$ is defined to be one in the *unweighted* case and is defined to the distance between the two vertices of the edge in the *weighted* case. An L_2 or L_1 unit-disk graph can be defined if the distance between points in measured by the L_2 or L_1 metric. We consider the following *reverse shortest path* (RSP) problem. In addition to P, given a value $\lambda > 0$ and two points $s, t \in P$, the problem is to compute the smallest value r such that the distance between s and t in $G_r(P)$ is at most λ. There are four cases for the RSP problem depending on whether L_1 or

L_2 metric is considered and whether the unit-disk graphs are weighted or not.

The L_1 distance selection problem in the plane can be solved in $O(n \log^2 n)$ time [16]. Therefore, one can perform a binary search in the set of all pairwise L_1 distances among n points of P using the algorithm in [16] as well as the corresponding decision algorithm (i.e., the L_1 SSSP algorithm) to solve the L_1 RSP problem for both the unweighted and weighted cases in $O(n \log^3 n)$ time. Note that the time complexity is dominated by the L_1 distance selection algorithm. We focus on the L_2 RSP problem in this dissertation.

Cabello and Jejčič [6] mentioned a straightforward solution that can compute the optimal parameter r^* in $O(n^{4/3} \log^3 n)$ time for both the unweighted and the weighted cases in the L_2 metric. We gave two algorithms for the L_2 unweighted case and their time complexities are $O(\lfloor \lambda \rfloor \cdot n \log n)$ and $O(n^{5/4} \log^{7/4} n)$, respectively; we also gave an algorithm of $O(n^{5/4} \log^{5/2} n)$ time for the L_2 weighted case.

Our results on this problem have been published in two conferences [17, 18] and one journal [19]. In particular, our paper [18] received the Best Student Paper Award of the conference. Refer to Chapter 3 for the details.

1.2.3 Computing the minimum bottleneck moving spanning tree

Given a set P of n moving points in the plane. 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; 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 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 a minimum bottleneck.

Previously, this problem was solved in $O(n^2)$ time by Akitaya, Biniaz, Bose, De Carufel, Maheshwari, Silveira, and Smid [20]. We present an algorithm of $O(n^{4/3}\log^3 n)$ time to compute T^*.

Our results on this problem have been published in a conference [21]. Refer to Chapter 4 for the details.

1.2.4 A simple algorithm for unit-disk range reporting

Given a set P of n points in the plane and a value r, we consider the following *unit-disk range reporting* problem (or UDRR for short): Construct a data structure such that given any query disk of radius r, all points of P in the disk can be reported efficiently.

The UDRR problem is also known as the *fixed-radius neighbor problem* in the literature [22–25]. Combining the framework in [26] with the shallow cutting algorithm [27], one can build a data structure of $O(n)$ space in $O(n\log n)$ deterministic time that can answer each UDRR query in $O(k+\log n)$ time, where k is the output size; note that this result also works for query disks of arbitrary radii.

We present a new UDRR data structure with the same complexity as above by exploiting special properties of unit disks. Our algorithm is much simpler than the algorithm of [26,27]. Indeed, the algorithm of [26,27] involves relatively advanced geometric techniques like shallow partition theorem and shallow cuttings in 3D, planar graph separators, computing ϵ-net and ϵ-approximations, etc. Our algorithm only relies on elementary techniques (the most complicated one might be a fractional cascading data structure [28,29]).

Our result on this problem has been submitted to a conference and is still under review. Refer to Chapter 5 for the details.

1.2.5 Improved algorithms for distance selection and related problems

We first consider the *distance selection* problem: Given a set P of n points in the plane and an integer $1 \le k \le \binom{n}{2}$, the problem asks for the k-th smallest interpoint distance among all pairs of points of P. The problem can be easily solved in $O(n^2)$ time. Katz and Sharir [30]

presented a deterministic algorithm that runs in $O(n^{4/3}\log^2 n)$ time. Very recently, Chan and Zheng proposed a randomized algorithm of $O(n^{4/3})$ expected time (see the arXiv version of [31]). Also, the time complexity can be made as a function of k. In particular, Chan's randomized techniques [32] solved the problem in $O(n\log n + n^{2/3}k^{1/3}\log^{5/3} n)$ expected time and Wang [33] recently improved the algorithm to $O(n\log n + n^{2/3}k^{1/3}\log n)$ expected time; these algorithms are particularly interesting when k is relatively small.

We first introduce a technique that improves the *partial batched range searching problem*. Then we present a new deterministic algorithm that solves the distance selection problem in $O(n^{4/3}\log n)$ time. Albeit slower than the randomized algorithm of Chan and Zheng [31], our algorithm is the first progress on the deterministic solution since the work of Katz and Sharir [30] published 25 years ago (30 years if we consider their conference version in SoCG 1993).

By extending and summarizing all techniques we used, we propose a general algorithmic framework that can be used to solve any geometric optimization problems that involve interpoint distances of a set of points in the plane. One application of this new framework is the *two-sided discrete Fréchet distance with shortcuts* problem, or *two-sided DFD* for short. Fréchet distance is used to measure the similarity between two curves and many of its variations have been studied, e.g., [34–39]. Avraham, Filtser, Kaplan, Katz, and Sharir [36] solved the two-sided DFD in $O((m^{2/3}n^{2/3}+m+n)\log^3(m+n))$, where m and n are the numbers of vertices of the two input curves, respectively. Using our new framework, we improve their algorithm to $O((m^{2/3}n^{2/3}\cdot 2^{O(\log^*(m+n))} + m\log n + n\log m)\log(m+n))$ time, an improvement of roughly $O(\log^2(m+n))$.

For the computation of the *one-sided discrete Fréchet distance with shortcuts* (*one-sided DFD* for short), the authors [36] gave a randomized algorithm of $O((m+n)^{6/5+\epsilon})$ expected time, for any constant $\epsilon > 0$. Using our techniques, we improve their algorithm to $O((m+n)^{6/5}\log^{8/5}(m+n))$ expected time. Based on the techniques of [36], Katz and Sharir [40] proposed a randomized algorithmic framework for solving geometric optimization problems that involve interpoint distances in a point set. We improve the framework to

$O(n^{4/3}/L^{1/3} \cdot \log^2 n + T_D \cdot \log n \cdot \log\log n + L^{1/2} \cdot T_D \cdot \log n)$ expected time, where T_D is the running time of the decision algorithm. Our result for the one-sided DFD is a direct application of this framework.

We also demonstrate that the randomized algorithms for the reverse shortest paths in unit-disk graphs for both unweighted and weighted cases can be improved over the previous work by using our new framework. Deterministic algorithms of $O(n^{4/3}\log^{7/4} n)$ and $O(n^{4/3}\log^{5/2} n)$ times are known for the unweighted and weighted problems, respectively [17, 18]. Katz and Sharir [40] solved both problems in $O(n^{6/5+\epsilon})$ expected time. With our improvement to the framework, we can now solve the unweighted problem in $O(n^{6/5}\log^{8/5} n)$ expected time and solve the weighted case in $O(n^{6/5}\log^{12/5} n)$ expected time.

Our result on these problems has been submitted to a conference and is still under review. Refer to Chapter 6 for the details.

1.3 Dissertation outline

The rest of this dissertation is organized as follows. We present our optimal shortest path algorithm for the L_1 unit-disk graph in Chapter 2. In Chapter 3, we give the results on the reverse shortest path problem for unit-disk graphs. Chapter 4 presents our algorithm for the Euclidean minimum bottleneck moving spanning tree. Our algorithm for unit-disk range reporting is introduced in Chapter 5. Chapter 6 provides our new distance selection algorithm as well as our algorithmic frameworks along with their applications. Finally, the future work is discussed in Chapter 7.

CHAPTER 2

L_1 SHORTEST PATHS IN UNIT-DISK GRAPHS

2.1 Introduction

We consider the problem of computing shortest paths from a given vertex to all other vertices in unit-disk graphs while the distance is measured by the L_1 metric. The results in this chapter have been published in a conference [14] and a journal [15].

2.1.1 Problem definitions and our results

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

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

The L_2 case of the problem where the distance is measured under the L_2 metric has been extensively studied [1, 6–10]. The current best algorithm, which was given by Wang and Xue [1], runs in $O(n \log^2 n)$ time. The L_1 case, however, has not been particularly studied before. To solve the L_1 problem, we follow the algorithmic framework of Wang and Xue [1] but give a faster implementation. The runtime of Wang and Xue's algorithm [1] is dominated by a bottleneck subproblem. Due to some special properties of the L_1 metric, we derive a more efficient algorithm for the bottleneck subproblem in L_1 case, which leads to an overall $O(n \log n)$-time algorithm for the shortest path problem.

More specifically, the bottleneck subproblem is the offline insertion-only additively-weighted nearest-neighbor problem, where we are given an offline sequence of k insertions and queries such that an *insertion* inserts a weighted point to a point set U (which is $\emptyset$ initially) and a *query* asks for the additively-weighted nearest neighbor in U of a query point. The goal is to answer all queries. Wang and Xue [1] solved the problem in $O(k \log^2 k)$ time by using the standard logarithmic method [41,42]. This leads to the overall $O(n \log^2 n)$ time for their shortest path algorithm [1]; reducing the time for the subproblem to $O(k \log k)$ would solve the shortest path problem in $O(n \log n)$ time. The difficulty in doing so is that there does not exist a semi-dynamic (for insertions only) weighted Voronoi diagram data structure that can perform each insertion in $O(\log k)$ amortized time (in order to answer queries, an efficient dynamic point location data structure is also needed). For solving our L_1 shortest path problem, we first observe that in the bottleneck subproblem U and V are separated by an axis-parallel line ℓ, where V is the set of all query points. Without loss of generality, we assume that ℓ is horizontal and U is below ℓ. Based on the properties of the L_1 metric, a critical observation we find is that the portion of the weighted L_1 Voronoi diagram of U above ℓ only consists of a set of vertical lines. Then, we can easily maintain these vertical lines by a balanced binary search tree so that each query can be answered in $O(\log k)$ time. Further, the special structure also allows us to update the portion of the Voronoi diagram above ℓ in $O(\log k)$ amortized time for each insertion. As such, the bottleneck subproblem can be solved in $O(k \log k)$ time in the L_1 case, which leads to an overall $O(n \log n)$ time algorithm for the shortest path problem. Note that the space of our shortest path algorithm is $O(n)$.

Cabello and Jejčič [6] observed that by a simple reduction from the max-gap problem, deciding whether the unit-disk graph $G(P)$ is connected requires $\Omega(n \log n)$ time even if all points of P are on a line. This implies that $\Omega(n \log n)$ is a lower bound for solving the shortest path problem in unit-disk graphs for both the L_1 and L_2 cases (because both cases are the same when all points of P are on a line). As such, our algorithm for the L_1 case is optimal.

2.1.2 Related work

Before Wang and Xue's work [1], the shortest path problem in the L_2 case had been studied by many others. Roditty and Segal [10] gave the first sub-quadratic algorithm of $O(n^{4/3+\epsilon})$ time for any constant $\epsilon > 0$. Cabello and Jejčič [6] later proposed an improved algorithm of $O(n^{1+\epsilon})$ time. Following the framework of Cabello and Jejčič [6] but with a more efficient data structure for the bichromatic closest pair problem, Kaplan et al. [9] gave a randomized algorithm that solves the problem in $O(n \log^{12+o(1)} n)$ expected time. Approximation algorithms for the problem have also been developed, e.g., see [1, 7, 8]

The shortest path problem we consider is actually on a *weighted* unit-disk graph. In the *unweighted* case, the weight of each edge of the graph is 1. The unweighted problem is much easier. The L_2 unweighted problem can be solved in $O(n \log n)$ time [6, 7]. In particular, if all input points of P are presorted by their x- and y- coordinates, the algorithm of Chan and Skrepetos [13] runs in $O(n)$ time.

As an important class of geometric intersection graphs, unit-disk graphs have been widely studied due to many of their applications, e.g., in wireless sensor networks [4, 5]. In addition to the shortest path problem, many other problems on unit-disk graphs have also been considered in the literature, such as the clique problem [11], the independent set problem [12], all pairs of shortest paths [7, 8, 13], the diameter problem [7, 8, 13], etc. Comparing to general graphs, these problems in unit-disk graphs can be solved more efficiently by exploiting their underlying geometric structures.

Outline. In the rest of this chapter, we describe the main algorithm in Section 2.2 while the bottleneck subproblem is tackled in Section 2.3.

2.2 The main algorithm

In this section, we describe the main algorithm for the shortest path problem. Our algorithm follows Wang and Xue's algorithmic framework [1]. In the following, we will adapt their algorithm to the L_1 case. We will also borrow some of their notation.

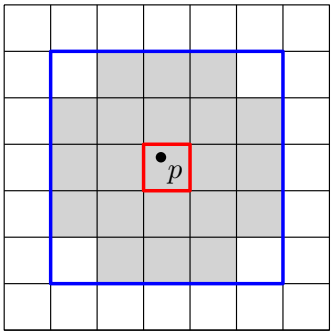


Fig. 2.1: The side length of each square cell in the grid Γ is $\frac{1}{2}$. For the black point p, the red cell that contains it is $\square_p$, and the square area bounded by blue segments which contains 5×5 cells is the patch $\boxplus_p$. For any point in $\square_p$, its neighboring points in $G(P)$ must lie in the grey region.

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

The algorithm follows the basic idea of Dijkstra's shortest path algorithm with the help of a grid. At the outset, we implicitly build a grid Γ of square cells of side length $1/2$. For simplicity of discussion, we assume that each vertex of $G(P)$ lies in the interior of a single cell of Γ. A *patch* of Γ is a square area consisting of 5×5 cells of Γ. For any point p in the plane, let $\square_p$ denote the cell of Γ that contains p and $\boxplus_p$ denote the patch whose central cell is $\square_p$ (see Fig. 2.1). Since the side length of each cell of Γ is $1/2$, if two vertices of $G(P)$ are in a single cell of Γ, they must be connected by an edge in $G(P)$. On the other hand, if two points p and q are connected by an edge in $G(P)$, then q must be in a cell of $\boxplus_p$. Unlike Dijkstra's shortest path algorithm, which selects one single vertex in each iteration to compute shortest-path information, our algorithm tries to compute shortest-path information for all vertices in a cell of Γ and then pass shortest-path information to the vertices in the neighboring cells.

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

To implicitly compute the grid Γ, we actually perform the following preprocessing. We

compute $P_\square$ for all cells $\square$ of Γ that contain at least one point of P. We also associate pointers to each point $p \in P$ such that from p we can access $\square_p$ and $\boxplus_p$. The preprocessing can be done in $O(n \log n)$ time and $O(n)$ space [1].

The algorithm will compute a table $dist[\cdot]$ for all vertices of $G(P)$, where $dist[p]$ is the length of a shortest path between s and a point $p \in P$. Note that we should also maintain the corresponding path-predecessor information to form a shortest path tree; this can be done by standard techniques [1], so we omit the discussions.

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

With $\textsc{Update}(U, V)$, the algorithm works as follows (refer to Algorithm 2.1 for the pseudocode). Initially, for each vertex $p \in P$, $dist[p]$ is set to ∞, except that $dist[s] = 0$. Initialize $Q = P$. In the main loop, as long as $Q \neq \emptyset$, in each iteration we find a vertex $q \in Q$ who has a minimum $dist[q]$. Subsequently there are two subroutines $\textsc{Update}(Q_{\boxplus_q}, Q_{\square_q})$ and $\textsc{Update}(Q_{\square_q}, Q_{\boxplus_q})$. Finally, vertices in $Q_{\square_q}$ are removed from Q, because $dist[p]$ for all $p \in Q_{\square_q}$ have been correctly computed. Refer to [1] for the correctness proof, which is applicable to the L_1 case.

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

The first update. For the first update $\textsc{Update}(Q_{\boxplus_q}, Q_{\square_q})$, the key is to find a point $q_v \in Q_{\boxplus_q} \cap \bigodot_v$ that minimizes $dist[q_v] + d(q_v, v)$ for each point $v \in Q_{\square_q}$. If we assign each point in $Q_{\boxplus_q}$ a weight equal to its $dist$-value, then q_v is essentially the additively-weighted nearest neighbor of v in $Q_{\boxplus_q} \cap \bigodot_v$. To find q_v efficiently, a crucial observation found by Wang and Xue [1] (see Lemma 2.5 in [1], whose proof is applicable to the L_1 case) is that any point $p \in Q_{\boxplus_q}$ that minimizes $dist[p] + d(p, v)$ must be in $\bigodot_v$, i.e., the nearest neighbor of v in $Q_{\boxplus_q}$ is also the nearest neighbor of v in $Q_{\boxplus_q} \cap \bigodot_v$. Due to this observation, we can find q_v for all $v \in Q_{\square_q}$ as follows. First, we build an L_1 additively-weighted Voronoi

Algorithm 2.1: The SSSP Algorithm [1]

```
1  Function SSSP(P, s):
2      for each p ∈ P do
3          dist[p] = ∞
4      end
5      dist[s] = 0
6      Q = P
7      while Q ≠ ∅ do
8          q = arg min_{p∈Q}{dist[p]}
9          UPDATE(Q_{⊞_q}, Q_{□_q}) // first update
10         UPDATE(Q_{□_q}, Q_{⊞_q}) // second update
11         Q = Q \ Q_{□_q}
12     end
13     return dist[·]
14 end
```

diagram on vertices in $Q_{\boxplus_q}$ and then using the diagram to find the nearest neighbor for each $v \in Q_{\square_q}$. Constructing the diagram can be done in $O(|Q_{\boxplus_q}| \log |Q_{\boxplus_q}|)$ time and $O(|Q_{\boxplus_q}|)$ space (e.g., by using the abstract Voronoi diagram algorithm [43]), and all queries together take $O(|Q_{\square_q}| \log |Q_{\boxplus_q}|)$ time (e.g., build a point location data structure on the diagram in $O(|Q_{\boxplus_q}|)$ time [44,45] and then perform point location queries for points of $Q_{\square_q}$, which take $O(\log |Q_{\boxplus_q}|)$ time each).

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

If $\square$ is $\square_q$, then $Q_{\square_q} = Q_{\square}$. Since the distance between any two points in $\square_q$ is at most 1, we can use the following algorithm to implement $\textsc{Update}(Q_{\square_q}, Q_{\square})$. We first build an L_1 weighted Voronoi diagram on points of $Q_{\square_q}$ in $O(|Q_{\square_q}| \log |Q_{\square_q}|)$ time and $O(|Q_{\square_q}|)$ space [43], and then use it to find the weighted nearest neighbor q_v for each point $v \in Q_{\square_q}$. Clearly, the total time is $O(|Q_{\square_q}| \log |Q_{\square_q}|)$.

If $\square$ is not $\square_q$, then a critical property is that $\square$ and $\square_q$ are separated by an axis-parallel line ℓ. To perform $\textsc{Update}(Q_{\square_q}, Q_{\square})$, Wang and Xue [1] proposed the following approach (see Algorithm 2.2 for the pseudocode). Let $U = Q_{\square_q}$ and $V = Q_{\square}$.

Algorithm 2.2: UPDATE(U, V) [1]

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

We first sort vertices in $U = \{u_1, u_2, ..., u_{|U|}\}$ by their *dist*-values such that $dist[u_1] \leq dist[u_2] \leq ... \leq dist[u_{|U|}]$. Then we partition V into subsets $V_i = \{v \in V \mid v \in \bigodot_{u_i}, v \notin \bigodot_{u_j} \text{ for all } j < i\}$, for all $i = 1, 2, \ldots, |U|$. For each $1 \leq i \leq |U|$, for each vertex $v \in V_i$, we find $q_v = \arg\min_{p \in U_i}\{dist[p] + d(p, v)\}$, where $U_i = \{u_i, u_{i+1}, \ldots, u_{|U|}\}$, and update $dist[v] = \min\{dist[v], dist[q_v] + d(q_v, v)\}$. This step is implemented by a for loop (Lines 6–13) in Algorithm 2.2. By the definition of V_i, we have $U \cap \bigodot_v \subseteq U_i$ for all $v \in V_i$. Also, Wang and Xue [1] proved that q_v found as above must be in $\bigodot_v$ (see Lemma 2.6 in [1], whose proof is applicable to the L_1 case). As such, $q_v = \arg\min_{p \in U \cap \bigodot_v}\{dist[p] + d(p, v)\}$. This proves the correctness of the algorithm.

We now analyze the runtime of the above algorithm. Sorting the vertices of U takes $O(|U| \log |U|)$ time. To compute the subsets V_i, $1 \leq i \leq |U|$, Wang and Xue [1] gave an algorithm of $O(k \log k)$ time (and $O(k)$ space) for the L_2 case (see Section 2.2.1 [1]) by making use of the property that U and V are separated by ℓ, where $k = |U| + |V|$. For the L_1 case, we can use the same algorithm; in fact, the algorithm becomes easier as a disk in the L_1 case is a diamond. We omit the details and conclude that the subsets V_i, $1 \leq i \leq |U|$, can be computed in $O(k \log k)$ time in the L_1 case. Next, the for loop (Lines 6–13) is for the bottleneck subproblem mentioned in Section 2.1, i.e., the offline insertion-only additively-

weighted nearest-neighbor problem. Indeed, if we assign each vertex in U a weight equal to its *dist*-value, then q_v is essentially the additively-weighted nearest neighbor of v in U', where $U' = U_i$ in the i-th iteration of the for loop. The set U' is dynamically changed with point insertions. Using the standard logarithmic method [41, 42], Wang and Xue [1] solves the problem in $O(k \log^2 k)$ time. By exploring the properties of the L_1 metric, we give an $O(k \log k)$ time (and $O(k)$ space) algorithm in Section 2.3. As such, UPDATE$(Q_{\square_q}, Q_{\square})$ can be performed in $O(k \log k)$ time and $O(k)$ space, with $k = |Q_{\square_q}| + |Q_{\square}|$.

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

Theorem 2.1. *Given a set P of n points in the L_1 plane and a source point $s \in P$, the shortest paths from s to all vertices in the unit-disk graph $G(P)$ can be computed in $O(n \log n)$ time and $O(n)$ space.*

Proof. As discussed before, constructing the grid Γ implicitly can be done in $O(n \log n)$ time and $O(n)$ space [1]. We have shown that both UPDATE procedures can be implemented in $O(|Q_{\boxplus_q}| \log |Q_{\boxplus_q}|)$ time and $O(|Q_{\boxplus_q}|)$ space. As such, each iteration of the while loop of Algorithm 2.1 can be implemented in $O(|Q_{\boxplus_q}| \log |Q_{\boxplus_q}|)$ time and $O(|Q_{\boxplus_q}|)$ space. As $\sum_{q \in Q} |Q_{\boxplus_q}| \leq 25n$, the total time of the algorithm is $O(n \log n)$. Note that the overall time of Line 8 and Line 11 of Algorithm 2.1 can be easily bounded by $O(n \log n)$ by using a balanced binary search tree. The total space of the algorithm is $O(n)$. □

2.3 The bottleneck subproblem

In this section, we present an $O(k \log k)$ time and $O(k)$ space algorithm to solve the bottleneck subproblem on U and V, with $k = |U| + |V|$. Recall U and V are separated by an axis-parallel line ℓ. Without loss of generality, we assume that ℓ is horizontal such that U is below ℓ and V is above ℓ. Our goal is to find $q_v \in U'$ for all $v \in V_i$ (i.e., Line 10 in Algorithm 2.2), for a subset $U' \subseteq U$.

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

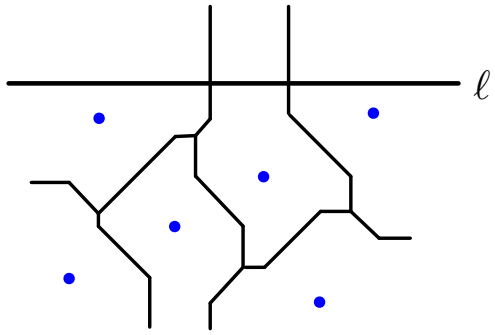


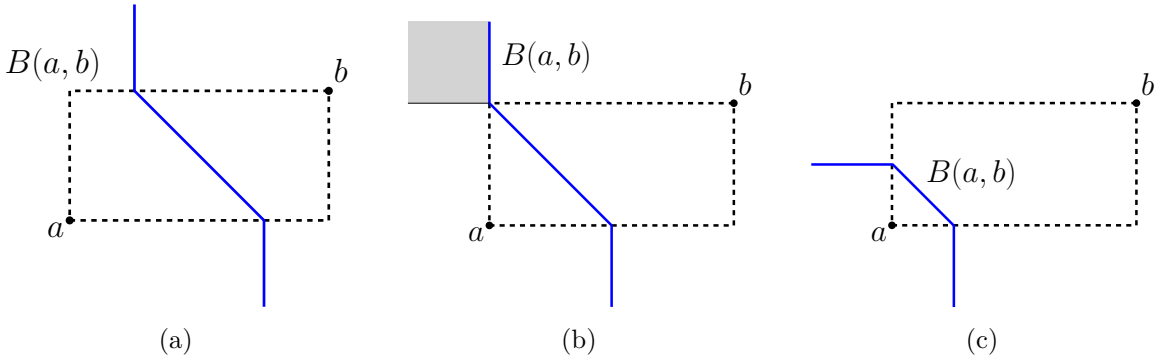
Fig. 2.2: Illustrating $VD(U')$, where U' has six blue points (with the same weight). $VD_h(U')$ consists of two vertical half-lines.

2.3.1 Observations

Let $VD(U')$ denote the weighted Voronoi diagram of U'. To find q_v, it suffices to locate the cell of $VD(U')$ that contains v. Let h denote the upper half-plane bounded by ℓ. As v is above ℓ, it suffices to maintain the portion of $VD(U')$ above ℓ, denoted by $VD_h(U')$. In what follows, we first show that $VD_h(U')$ has a very simple structure: it only consists of a set of vertical half-lines with endpoints on ℓ and going upwards to the infinity (e.g., see Fig. 2.2). Then, we will show that $VD_h(U')$ can be updated in $O(\log k)$ amortized time for each insertion (i.e., inserting a point into U').

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

Properties of bisectors of two weighted points. Consider two weighted points a and b in the plane with nonnegative weights $w(a)$ and $w(b)$, respectively. The *bisector* $B(a, b)$ of a and b is the locus of points with equal (additively-)weighted distance to a and b, i.e., $B(a, b) = \{p \in \mathbb{R}^2 \mid w(a) + d(a, p) = w(b) + d(b, p)\}$ (e.g., see Fig. 2.3). Note that in the degenerate case it is possible that an entire quadrant of the plane is in $B(a, b)$ (e.g., see Fig. 2.3b), in which case we only consider the vertical boundary of the quadrant to be in $B(a, b)$. Hence, $B(a, b)$ in general consists of three parts: two axis-parallel half-lines with a segment in the middle. Suppose both a and b are below the line ℓ and $x(a) \leq x(b)$. Define $B_h(a, b) = B(a, b) \cap h$. Then either $B_h(a, b) = \emptyset$ or $B_h(a, b) \cap h$ is a vertical half-line



(a) (b) (c)

Fig. 2.3: Possible cases for the bisector $B(a, b)$ of two weighted points a and b.

grounded on ℓ; in the latter case $x(a) \leq x(B_h(a, b)) \leq x(b)$. Note that if $x(a) = x(b)$, then $B(a, b)$ is a horizontal line between a and b and thus $B_h(a, b) = \emptyset$.

Geometric structure of $VD_h(U')$. Since all points of U are below ℓ, according to the discussion above, for any two points u_i and u_j of U, $B_h(u_i, u_j)$ is either $\emptyset$ or a vertical half-line grounded on ℓ (and the vertical half-line is between u_i and u_j). These properties guarantee that $VD_h(U')$ consists of a set of $O(|U'|)$ vertical half-lines grounded on ℓ (e.g., see Fig. 2.2), and between each pair of adjacent half-lines is the portion of the Voronoi cell of a vertex $u \in U'$. As such, we can use a balanced binary search tree $T(U')$ to store the x-coordinates of the vertical half-lines of $VD_h(U')$. Given a query point $v \in V$, we can use $T(U')$ to find the cell of $VD_h(U')$ containing v and thus obtain q_v in $O(\log |U'|)$ time, which is $O(\log k)$ as $|U'| \leq |U| \leq k$. In the following, we will discuss how to update $VD_h(U')$ after a point of U is inserted to U'. We first prove some properties about the geometric structure of $VD_h(U')$.

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

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

will be useful for processing insertions.

Lemma 2.1. *Suppose $u^1, u^2, \ldots, u^t$ is the list of relevant vertices of U' whose Voronoi cells intersect h in the order from left to right. Then, the followings hold.*

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

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

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

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

The fourth lemma statement is an immediate consequence of the third lemma statement. □

2.3.2 Processing insertions

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

Consider inserting a point $u^* \in U \setminus U'$ into U'. As $u^* \in U$, u^* is below ℓ. Let $U'' = U' \cup \{u^*\}$. Our goal is to construct $VD_h(U'')$ by modifying $VD_h(U')$, or more precisely, obtain the tree $T(U'')$ by modifying $T(U')$. For differentiation, for each vertex $u \in U''$, we

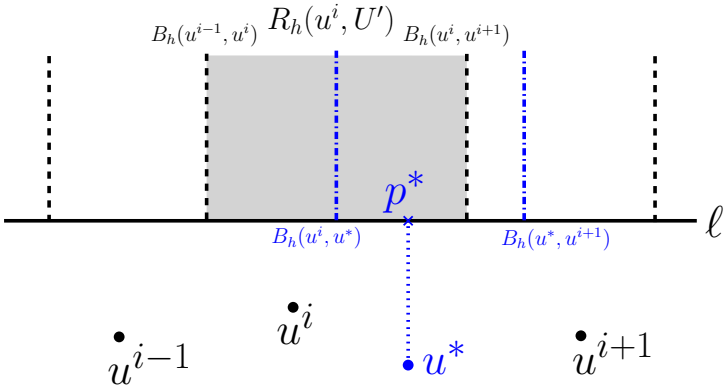


Fig. 2.4: Illustrating $VD_h(U')$, and $VD_h(U'')$ after u^* is inserted. The two dash dotted blue segments are new half-lines in $VD_h(U'')$ while $B_h(u^i, u^{i+1})$ does not appear in $VD_h(U'')$. $R_h(u^i, U')$ is the grey area and $R_h(u^*, U'')$ is the region between the two dash dotted blue segments. Note that $B_h(u^{i-1}, u^i)$ is $l_{u^i} = r_{u^{i-1}}$ and $B_h(u^i, u^{i+1})$ is $r_{u^i} = l_{u^{i+1}}$.

use $R(u, U'')$ to denote the Voronoi cell of u in $VD(U'')$ and use $R(u, U')$ to denote the Voronoi cell of u in $VD(U')$. We define $R_h(u, U'')$ and $R_h(u, U')$ similarly. Let $u^1, u^2, \ldots, u^t$ be the list of relevant vertices of U' whose Voronoi cells intersect h ordered from left to right.

We first compute the vertical projection of u^* on ℓ and let p^* denote the projection point (e.g., see Fig. 2.4). Then, using the tree $T(U')$, we find the cell $R_h(u^i, U')$ of $VD_h(U')$ that contains p^*, for some relevant point $u^i \in U'$. For ease of discussion, we assume $1 < i < t$ and other cases can be handled similarly. The following lemma is obtained based on Lemma 2.1.

Lemma 2.2. *$R_h(u^*, U'') \neq \emptyset$ if and only if $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$, and if $R_h(u^*, U'') \neq \emptyset$, then $p^* \in R_h(u^*, U'')$.*

Proof. If $R_h(u^*, U'') \neq \emptyset$, then by Lemma 2.1, p^* must be in $R_h(u^*, U'')$ and this implies $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$ must hold. On the other hand, suppose $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$. Then, since $p^* \in R_h(u^i, U')$, $d(p^*, u^i) + w(u^i) \leq d(p^*, u) + w(u)$ holds for any vertex $u \in U'$. Therefore, $d(p^*, u) + w(u) \geq d(p^*, u^*) + w(u^*)$ holds for any $u \in U''$. This implies that u^* is the nearest neighbor of p^* in U''. As such, the point p^* must be in $R_h(u^*, U'')$ and $R_h(u^*, U'')$ cannot be empty. □

With Lemma 2.2, our insertion algorithm proceeds as follows. We check whether $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$. If not, then $R_h(u^*, U'') = \emptyset$ by Lemma 2.2 and thus $VD_h(U'') = VD_h(U')$; hence, $T(U'') = T(U')$ and we are done with processing the insertion of u^*. In the following, we assume that $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$. By Lemma 2.2, $R_h(u^*, U'') \neq \emptyset$ and thus $VD_h(U'') \neq VD_h(U')$. Below we discuss how to modify $VD_h(U')$ to obtain $VD_h(U'')$.

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

Since $p^* \in R_h(u^i, U')$, we have $x(u^*) = x(p^*) \in [x(l_{u^i}), x(r_{u^i})]$. By Lemma 2.1, $x(u^{i-1}) \leq x(r_{u^{i-1}}) = x(l_{u^i})$ and $x(r_{u^i}) = x(l_{u^{i+1}}) \leq x(u^{i+1})$. Therefore, $x(p^*) \in [x(u^{i-1}), x(u^{i+1})]$. Also by Lemma 2.1, $x(u^{i-1}) < x(u^i) < x(u^{i+1})$. Without loss of generality, we assume that $x(u^i) \leq x(p^*) < x(u^{i+1})$. We first discuss how to obtain the portion of $VD_h(U'')$ to the left of p^*. To this end, we consider the points $u^i, u^{i-1}, \ldots, u^1$ in this order.

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

- If $B_h(u^i, u^*) \neq \emptyset$, then $B_h(u^i, u^*)$ is a vertical half-line grounded on ℓ. Since $x(u^i) \leq x(u^*)$, according to the properties of bisectors, $x(u^i) \leq x(B_h(u^i, u^*)) \leq x(u^*)$. As $x(l_{u^i}) \leq x(u^i)$ and $x(u^*) \leq x(r_{u^i})$, $B_h(u^i, u^*)$ must be in the Voronoi cell $R_h(u^i, U')$ between l_{u^i} and p^* (e.g., see Fig. 2.4). Hence, $B_h(u^i, u^*)$ must be the right bounding half-line of the cell $R_h(u^i, U'')$ in $VD_h(U'')$ as well as the left bounding half-line of the cell $R_h(u^*, U'')$. We update the tree $T(U')$ accordingly (i.e., insert $B_h(u^i, u^*)$ to $T(U')$) and then halt the algorithm (i.e., the construction of $VD_h(U'')$ on the left of p^* is finished).

- If $B_h(u^i, u^*) = \emptyset$, then by our definition of bisectors (including our way for handling the degenerating case), since $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$, $d(p, u^i) + w(u^i) \geq d(p, u^*) + w(u^*)$ holds for any point $p \in h$. This implies that u^i is *dominated* by u^* with respect to the points of h, and thus u^i becomes irrelevant in $VD_h(U'')$. As such, we remove l_{u^i} from $T(U')$. Note that l_{u^i} is $r_{u^{i-1}}$ by Lemma 2.2.

Next, we consider u^{i-1} in a way similar to the above for u^i. If $B_h(u^{i-1}, u^*) \neq \emptyset$, then $B_h(u^{i-1}, u^*)$ becomes the right bounding half-line of the cell $R_h(u^{i-1}, U'')$ in $VD_h(U'')$ as well as the left bounding half-line of $R_h(u^*, U'')$. We insert $B_h(u^{i-1}, u^*)$ into $T(U')$ and halt the algorithm. If $B_h(u^{i-1}, u^*) = \emptyset$, then since $p^* \in R_h(u^*, U'')$ by Lemma 2.2, $d(p^*, u^{i-1}) + w(u^{i-1}) \geq d(p^*, u^*) + w(u^*)$. Further, by our definition of bisectors (including our way for handling the degenerating case), $d(p, u^{i-1}) + w(u^{i-1}) \geq d(p, u^*) + w(u^*)$ holds for any point $p \in h$. Therefore, as above, u^{i-1} becomes irrelevant in $VD_h(U'')$. Accordingly, we remove $l_{u^{i-1}}$ from $T(U')$. We then proceed to considering u^{i-2} in the same way as above.

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

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

Lemma 2.3. *After a point $u^* \in U$ is inserted into U', $VD_h(U'')$ can be computed from $VD_h(U')$ in $O((\delta + 1) \log k)$ time, where $U'' = U' \cup \{u^*\}$ and δ is the number of relevant vertices of $VD_h(U')$ that become irrelevant in $VD_h(U'')$.*

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

If $d(p^*, u^i) + w(u^i) < d(p^*, u^*) + w(u^*)$, then $VD_h(U'') = VD_h(U')$ by Lemma 2.2 and thus our algorithm is correct in this case. In the following, we assume that $d(p^*, u^i) + w(u^i) \geq d(p^*, u^*) + w(u^*)$ and prove that the diagram $VD_h(U'')$ constructed by our algorithm is correct.

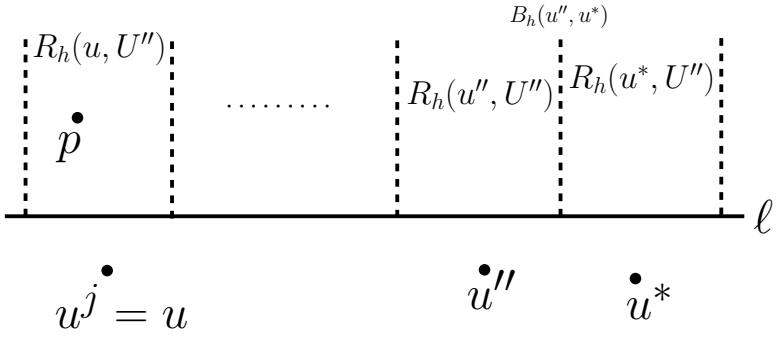


Fig. 2.5: Illustrating the proof of Lemma 2.3 for the case where u is not adjacent to u^* in L.

Let p be any point in h and let u be the point of U'' such that p is in the cell of u after our insertion algorithm for u^* is finished, i.e., $p \in R_h(u, U'')$. To prove the correctness of our algorithm, it suffices to show that $d(p, u) + w(u) \leq d(p, u') + w(u')$ holds for every point $u' \in U''$. Depending on whether $u = u^*$, there are two cases. Let u^j be the point of U' such that $p \in R_h(u^j, U')$.

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

- We then consider the case $u \neq u^*$. In this case, according to our algorithm, u must be u^j and u and u^* define different cells in $VD_h(U'')$, i.e., $R_h(u, U'') \neq R_h(u^*, U'')$. Without loss of generality, we assume that $R_h(u, U'')$ is to the left of $R_h(u^*, U'')$. Depending on whether u is adjacent to u^* in the relevant point list L after the insertion algorithm (L is defined in the same way as Lemma 2.1 with respect to $VD_h(U'')$), there are two subcases.

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

$d(p, u') + w(u')$ holds for any $u' \in U'$, we obtain that $d(p, u) + w(u) \leq d(p, u') + w(u')$ holds for any $u' \in U''$.

If u is not adjacent to u^* in L, then let u'' be the left neighboring relevant point of u^* in L (e.g., see Fig 2.5). Since $R_h(u, U'')$ is to the left of $R_h(u^*, U'')$ and $p \in R_h(u, U'')$, p must be to the left of $B_h(u'', u^*)$, which is the right bounding half-line of $R_h(u'', U'')$. As u'' is the left neighboring relevant point of u^* in L, according to our insertion algorithm, $d(p', u'') + w(u'') \leq d(p', u^*) + w(u^*)$ for any point $p' \in h$ to the left of $B_h(u'', u^*)$. Because p is in h to the left of $B_h(u'', u^*)$, $d(p, u'') + w(u'') \leq d(p, u^*) + w(u^*)$ holds. As $d(p, u^j) + w(u^j) \leq d(p, u') + w(u')$ for any $u' \in U'$, we have $d(p, u^j) + w(u^j) \leq d(p, u'') + w(u'')$. We thus derive $d(p, u^j) + w(u^j) \leq d(p, u^*) + w(u^*)$. Since $u = u^j$, we obtain that $d(p, u) + w(u) \leq d(p, u') + w(u')$ for any $u' \in U''$.

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

Note that once a relevant point becomes irrelevant after an insertion, it will never become relevant again for any insertions in future. Therefore, the total sum of δ in Lemma 2.3 for processing all insertions of U is at most k. As such, by Lemma 2.3, the total time for processing all insertions is $O(k \log k)$.

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

Lemma 2.4. *The bottleneck subproblem on U and V can be solved in $O(k \log k)$ time and $O(k)$ space, where $k = |U| + |V|$.*

CHAPTER 3

REVERSE SHORTEST PATH PROBLEM FOR UNIT-DISK GRAPHS

3.1 Introduction

We consider the reverse shortest path (RSP) problem for L_2 unit-disk graphs in both unweighted and weighted cases. The results in this chapter have been published in two conferences [17, 18] and one journal [19]. In particular, the paper [18] received the Best Student Paper Award from the conference.

3.1.1 Problem definitions and our results

Given a set P of n points in the plane and a parameter r, the *unit-disk graph* $G_r(P)$ is an undirected graph whose vertex set is P such that an edge connects two points $p, q \in P$ if the (Euclidean) distance between p and q is at most r. The weight of each edge of $G_r(P)$ is defined to be one in the *unweighted* case and is defined to be the distance between the two vertices of the edge in the *weighted* case. Alternatively, $G_r(P)$ can be viewed as the intersection graph of the set of congruent disks centered at the points of P with radii equal to $r/2$, i.e., two vertices are connected if their disks intersect. The *length* of a path in $G_r(P)$ is the sum of the weights of the edges of the path.

Computing shortest paths in unit-disk graphs with different distance metrics and different weights assigning methods has been extensively studied, e.g., [1,6–10,13]. Although a unit-disk graph may have $\Omega(n^2)$ edges, geometric properties allow to solve the single-source-shortest-path problem (SSSP) in sub-quadratic time. Roditty and Segal [10] first proposed an algorithm of $O(n^{4/3+\epsilon})$ time for unit-disk graphs for both unweighted and weighted cases, for any $\epsilon > 0$. Cabello and Jejčič [6] gave an algorithm of $O(n \log n)$ time for the unweighted case. Using a dynamic data structure for bichromatic closest pairs [46], they also solved the weighted case in $O(n^{1+\epsilon})$ time [6]. Chan and Skrepetos [13] gave an $O(n)$

time algorithm for the unweighted case, assuming that all points of P are presorted. Kaplan et al. [9] and Liu [47] developed new randomized results for the dynamic bichromatic closest pair problem; in particular, applying the result of Liu [47] to the algorithm of [6] leads to an $O(n \log^{9+o(1)} n)$ expected time randomized algorithm for the weighted case. Recently, Wang and Xue [1] proposed a new algorithm that solves the weighted case in $O(n \log^2 n)$ time. Some approximation algorithms for the problem have also been developed [1, 7, 8].

The L_1 version of the SSSP problem has also been studied, where the distance of two points in the plane is measured under the L_1 metric when defining $G_r(P)$. Note that in the L_1 version a "disk" is a diamond. The SSSP algorithms of [6, 13] for the L_2 unweighted version can be easily adapted to the L_1 unweighted version. Wang and Zhao [14] recently solved the L_1 weighted case in $O(n \log n)$ time. It is known that $\Omega(n \log n)$ is a lower bound for the SSSP problem in both L_1 and L_2 versions [6, 14]. Hence, the SSSP problem in the L_1 weighted/unweighted case as well as in the L_2 unweighted case has been solved optimally.

In this chapter, we consider the following *reverse shortest path* (RSP) problem. In addition to P, given a value $\lambda > 0$ and two points $s, t \in P$, the problem is to compute the smallest value r such that the distance between s and t in $G_r(P)$ is at most λ. There are four cases for the RSP problem depending on whether L_1 or L_2 metric is considered and whether the unit-disk graphs are weighted or not. Throughout the chapter, we let r^* denote the optimal value r for any case. The goal is therefore to compute r^*.

Observe that r^* must be equal to the distance of two points in P in any case (i.e., L_1, L_2, weighted, unweighted). In light of this observation, Cabello and Jejčič [6] mentioned a straightforward solution that can compute r^* in $O(n^{4/3} \log^3 n)$ time for both the unweighted and the weighted cases in the L_2 metric, by using the distance selection algorithm of Katz and Sharir [30] to perform binary search on all interpoint distances of P. In this chapter, we give two algorithms for the L_2 unweighted case and their time complexities are $O(\lfloor \lambda \rfloor \cdot n \log n)$ and $O(n^{5/4} \log^{7/4} n)$, respectively; we also give an algorithm of $O(n^{5/4} \log^{5/2} n)$ time for the L_2 weighted case.

The L_1 distance selection problem in the plane can be solved in $O(n \log^2 n)$ time [16].

Therefore, one can perform a binary search in the set of all pairwise L_1 distances among n points of P using the algorithm in [16] as well as the corresponding decision algorithm (i.e., the L_1 SSSP algorithm) to solve the L_1 RSP problem for both the unweighted and weighted cases in $O(n \log^3 n)$ time. Note that the time complexity is dominated by the L_1 distance selection algorithm. We focus on the L_2 RSP problem in this chapter.

Since the original reporting of our results,[1] some exciting progress has been made by Katz and Sharir [40], who proposed randomized algorithms of $O(n^{6/5+\epsilon})$ expected time for the L_2 RSP problem for both the unweighted and weighted cases, for any arbitrarily small $\epsilon > 0$. Note that all our results are deterministic.

Note that reverse/inverse shortest path problems have been studied in the literature under various problem settings. Roughly speaking, the problems are to modify the graph (e.g., modify some edge weights) so that certain desired constraints related to shortest paths in the graph can be satisfied, e.g., [48, 49]. Our reverse shortest path problem in unit-disk graphs may find applications in scenarios like the following. Consider $G_r(P)$ as an L_2 unit-disk intersection graph representing a wireless sensor network in which each disk represents a sensor and two sensors can communicate with each other (e.g., directly transmit a message) if there is an edge connecting them in $G_r(P)$. The disk radius is proportional to the energy of the sensor. For two specific sensors s and t, suppose we want to know the minimum energy for all sensors so that s and t can transmit messages to each other within λ steps for a given value λ. It is easy to see that this is equivalent to our L_2 RSP problem in the unweighted case. If the latency of transmitting a message between two neighboring sensors is proportional to their Euclidean distance and we want to know the minimum energy for all sensors so that the total latency of transmitting messages between s and t is no more than a target value λ, then the problem becomes the weighted case.

In addition to the shortest path problem, many other problems of unit-disk graphs have also been studied, i.e. clique [11], independent set [12], distance oracle [7, 8], diameter [7, 8,

[1] Our algorithms for the L_2 unweighted case were included in [17]; our results for the L_2 weighted case have been presented in the 29th Fall Workshop on Computational Geometry (FWCG 2021) and has also been accepted in [18]. Note that the second algorithm for the L_2 unweighted case runs in $O(n^{5/4} \log^2 n)$ time in [17]; in this full version, we slightly improve the time to $O(n^{5/4} \log^{7/4} n)$ by changing the threshold for defining large cells from $n^{3/4}$ to $(n/\log n)^{3/4}$ in Section 3.4.

13], etc. Comparing to general graphs, many problems can be solved efficiently in unit-disk graphs by exploiting their underlying geometric structures, although there are still problems that are NP-hard for unit-disk graphs and other geometric intersection graphs, e.g., [11,50].

3.1.2 Our approach

We present RSP algorithms for unit-disk graphs in the L_2 metric.

As the length of any path in $G_r(P)$ is an integer in the unweighted case, the length of a path of $G_r(P)$ is at most λ if and only if the length of the path is at most $\lfloor\lambda\rfloor$; therefore, we can replace λ in the unweighted problem by $\lfloor\lambda\rfloor$. In the following, we simply assume that λ is an integer in the unweighted case. Recall that our goal is to compute r^*, which must be equal to the distance of two points in P in both the unweighted and weighted cases. Given a value r, *the decision problem* is to decide whether $r \geq r^*$. It is not difficult to see that $r \geq r^*$ if and only if the distance of s and t in $G_r(P)$ is at most λ. Therefore, the decision problem can be solved efficiently by using the shortest path algorithm for the corresponding case [6, 13]. More specifically, with $O(n \log n)$-time preprocessing (to sort the points of P), given any r, whether $r \geq r^*$ can be decided in $O(n)$ time for the unweighted unit-disk graphs by the algorithm of Chan and Skrepetos [13]. For the weighted case, the decision problem can be solved in $O(n \log^2 n)$ time by Wang and Xue's shortest path algorithm [1].

Since r^* must be equal to the distance of two points of P, we can find r^* by doing binary search on the set of pairwise distances of all points of P. Given any $1 \leq k \leq \binom{n}{2}$, the distance selection algorithm of Katz and Sharir [30] can compute the k-th smallest distance among all pairs of points of P in $O(n^{4/3} \log^2 n)$ time. Using this algorithm, the binary search can find r^* in $O(n^{4/3} \log^3 n)$ time for both the unweighted and weighted cases. This is the algorithm mentioned in [6].

Our RSP algorithms are based on parametric search [51, 52], by parameterizing the decision algorithm of Chan and Skrepetos [13] (which we refer to as the CS algorithm) in the unweighted case, and parameterizing the decision algorithm of Wang and Xue [1] (which we refer to as the WX algorithm) in the weighted case. Below is an overview on our algorithms.

The unweighted case. The CS algorithm first builds a grid in the plane and then runs the breadth-first-search (BFS) algorithm with the help of the grid; in the i-th step of the BFS, the algorithm finds the set of points of P whose distances from s in $G_r(P)$ are equal to i. Although we do not know r^*, we run the CS algorithm on a parameter r in an interval $(r_1, r_2]$ such that each step of the algorithm behaves the same as the CS algorithm running on r^*. The algorithm terminates after t is reached, which will happen within λ steps. In each step, we use the CS algorithm to compare r^* with certain *critical values*, and the interval $(r_1, r_2]$ will be shrunk based on the results of these comparisons. Once the algorithm terminates, r^* is equal to r_2 of the current interval $(r_1, r_2]$. With the linear-time decision algorithm (i.e., the CS algorithm [13]), each step runs in $O(n \log n)$ time. The total time of the algorithm is $O(\lambda \cdot n \log n)$.

The above algorithm is only interesting when λ is relatively small. In the worst case, however, λ can be $\Theta(n)$, which would make the running time become $O(n^2 \log n)$. Next, by combining the strategies of the parametric search and the L_2 distance selection algorithm [30], we derive a better algorithm. The main idea is to partition the cells of the grid in the CS algorithm into two types: *large cells*, which contain at least $(n/\log n)^{3/4}$ points of P each, and *small cells* otherwise. For small cells, we process them using the above binary search algorithm with the L_2 distance selection algorithm [30]; for large cells, we process them using the above parametric search techniques. This works out due to the following observation. On the one hand, the number of large cells is relatively small (at most $O(n^{1/4} \log^{3/4} n)$) and thus the number of steps using the parametric search is also small. On the other hand, each small cell contains relatively few points of P (at most $O((n/\log n)^{3/4})$) and thus the total time we spend on the L_2 distance selection algorithm is not big. The threshold value $(n/\log n)^{3/4}$ is carefully chosen so that the total time for processing the two types of cells is minimized. In addition, instead of applying the L_2 distance selection algorithm [30] directly, we find that it suffices to use only a subroutine of that algorithm, which not only simplifies the algorithm but also reduces the total time by a logarithmic factor. All these efforts lead to an $O(n^{5/4} \log^{7/4} n)$ time algorithm to compute r^*.

The weighted case. Our algorithm for the L_2 weighted case also follows the parametric search scheme, by parameterizing the WX algorithm [1] instead. Like the unweighted case, we run the decision algorithm (i.e., the WX algorithm) with a parameter $r \in (r_1, r_2]$ by simulating the decision algorithm on the unknown r^*. At each step of the algorithm, we call the decision algorithm on certain critical values r to compare r and r^*, and the algorithm will proceed accordingly based on the result of the comparison. The interval $(r_1, r_2]$ will also be shrunk after these comparisons but is guaranteed to contain r^* throughout the algorithm. The algorithm terminates once the point t is reached, at which moment we can prove that r^* is equal to r_2 of the current interval $(r_1, r_2]$. The parametric search algorithm runs in $\Omega(n^2)$ time because t may be reached after $\Theta(n)$ steps. To further reduce the time, similarly to the L_2 unweighted case, we combine the strategies of the parametric search and the L_2 distance selection techniques [30]. The cells of the grid built in the algorithm are partitioned into large and small cells, but with a different threshold of $n^{3/4}\log^{3/2} n$. With this approach, the runtime of the algorithm can be bounded by $O(n^{5/4}\log^{5/2} n)$.

Outline. The rest of the chapter is organized as follows. Section 3.2 defines notation and reviews the CS algorithm. Our first algorithm for the unweighted case is presented in Section 3.3 while the second one is described in Section 3.4. Section 3.5 solves the weighted RSP problem. Section 3.6 concludes with remarks showing that our techniques can be readily extended to solve a more general "single-source" version of the RSP problem.

3.2 Preliminaries

Throughout the chapter, we will use "points of P" and "vertices of the graph $G_r(P)$" interchangeably. For any parameter r, let $d_r(p, q)$ denote the distance of two vertices p and q in $G_r(P)$. It is easy to see that $d_r(p, q) \leq d_{r'}(p, q)$ if $r \geq r'$.

For any two points p and q in the plane, let $\|p - q\|$ denote their Euclidean distance. For any subset P' of P and any region R in the plane, we use $P'(R)$ or $P' \cap R$ to refer to the subset of points P' contained in R. For any point p, let $x(p)$ and $y(p)$ denote its x- and y-coordinates, respectively.

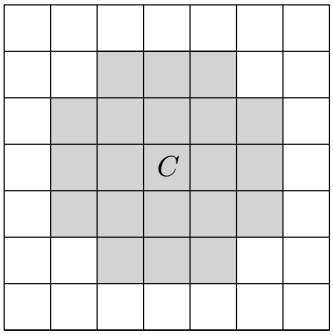


Fig. 3.1: The grey cells are all neighbor cells of C.

We next review the CS algorithm [13], which will help understand our RSP algorithms given later. Suppose we have a sorted list of P by x-coordinate and another sorted list of P by y-coordinate. Given a parameter r and a source point $s \in P$, the CS algorithm can compute in $O(n)$ time the distances from s to all other points of P in $G_r(P)$.

The first step is to compute a grid $\Psi_r(P)$ of square cells whose side lengths are $r/\sqrt{2}$. The grid technique was widely used in algorithms for unit-disk graphs [1, 13, 21]. A cell C' of $\Psi_r(P)$ is a *neighbor* of another cell C if the minimum distance between a point of C and a point of C' is at most r. Note that the number of neighbors of each cell of $\Psi_r(P)$ is $O(1)$ (e.g., see Fig. 3.1) and the distance between any two points in each cell is at most r.

Next, starting from the point s, the algorithm runs BFS in $G_r(P)$ with the help of the grid $\Psi_r(P)$. Define S_i as the subset of points of P whose distances in $G_r(P)$ from s are equal to i. Initially, $S_0 = \{s\}$. Given S_{i-1}, the i-th step of the BFS is to compute S_i by using S_{i-1} and the grid $\Psi_r(P)$, as follows. If a point p is not in $\bigcup_{j=0}^{i-1} S_j$, we say that p has not been *discovered* yet. For each cell C that contains at least one point of S_{i-1}, we need to find points that are not discovered yet and at distances at most r from the points of $S_{i-1} \cap C$ (i.e., the points of S_{i-1} in C); clearly, these points are either in C or in the neighbor cells of C. For points of $P(C)$, since every two points of C are within distance r from each other, we add all points of $P(C)$ that have not been discovered to S_i. For each neighbor cell C' of C, we need to solve the following subproblem: find the points of $P(C')$ that are not discovered yet and within distance at most r from the points of $S_{i-1} \cap C$. Since C' and C are separated by either a vertical line or a horizontal line, we essentially have the

following subproblem.

Subproblem 3.1. *Given a set of n_r red points below a horizontal line ℓ and a set of n_b blue points above ℓ, both sorted by x-coordinate, determine for each blue point whether there is a red point at distance at most r from it.*

The subproblem can be solved in $O(n_r + n_b)$ time as follows. For each red point p, the circle of radius r centered at p has at most one arc above ℓ (we say that this arc is defined by p). Let Γ be the set of these arcs defined by all red points. Since all arcs of Γ have the same radius and all red points are below ℓ, every two arcs intersect at most once and the arcs above ℓ are x-monotone. Further, as all red points are sorted already by x-coordinate, the upper envelope of Γ, denoted by $\mathcal{U}$, can be computed in $O(n_r)$ time by an algorithm similar in spirit to Graham's scan. Then, it suffices to determine whether each blue point is below $\mathcal{U}$, which can be done in $O(n_r + n_b)$ time by a linear scan. More specifically, we can first sort the vertices of $\mathcal{U}$ and all blue points. After that, for each blue point p, we know the arc of $\mathcal{U}$ that spans p (i.e., $x(p)$ is between the x-coordinates of the two endpoints of the arc), and thus we only need to check whether p is below the arc. In summary, solving the subproblem involves three subroutines: (1) compute $\mathcal{U}$; (2) sort all vertices of $\mathcal{U}$ with all blue points; (3) for each blue point p, determine whether it is below the arc of $\mathcal{U}$ that spans p.

The above computes the set S_i. Note that if $S_i = \emptyset$, then we can stop the algorithm because all points of P that can be reached from s in $G_r(P)$ have been computed. For the running time, notice that points of P in each cell of the grid $\Psi_r(P)$ can be involved in at most two steps of the BFS. Further, since each grid cell has $O(1)$ neighbors, the total time of the BFS algorithm is $O(n)$.

In order to achieve $O(n)$ time for the overall algorithm, the grid $\Psi_r(P)$ must be implicitly constructed. The CS algorithm [13] does not provide any details about that. There are various ways to do so. Below we present our method, which will facilitate our algorithm in the next section.

The grid $\Psi_r(P)$ we are going to build is a rectangle that is partitioned into square cells of side lengths $r/\sqrt{2}$ by $O(n)$ horizontal and vertical lines. These partition lines will be explicitly computed. Let P' be the subset of points of P located in $\Psi_r(P)$. P' has the following property: for each $p \in P \setminus P'$, p cannot be reached from s in $G_r(P)$, i.e., the distances from s to the points of $P \setminus P'$ in $G_r(P)$ are infinite. Let $\mathcal{C}$ denote the set of cells of $\Psi_r(P)$ that contain at least one point of P. For each cell $C \in \mathcal{C}$, let $N(C)$ denote the set of neighbors of C in $\mathcal{C}$. The information computed in the following lemma suffices for implementing the above BFS algorithm in linear time.

Lemma 3.1. *Suppose we have a sorted list of P by x-coordinate and another sorted list of P by y-coordinate. Both P' and $\mathcal{C}$, along with all vertical and horizontal partition lines of $\Psi_r(P)$, can be computed in $O(n)$ time. Further, with $O(n)$ time preprocessing, the following can be achieved:*

1. *Given any point $p \in P'$, the cell of $\mathcal{C}$ that contains p can be obtained in $O(1)$ time.*
2. *Given any cell $C \in \mathcal{C}$, the neighbor set $N(C)$ can be obtained in $O(|N(C)|)$ time.*
3. *Given any cell $C \in \mathcal{C}$, the subset $P(C)$ of P can be obtained in $O(|P(C)|)$ time.*

Proof. Let P_1 be the subset of P to the right of s including s. Let $s = p_1, p_2, \ldots, p_m$ be the list of P_1 sorted from left to right, with $m = |P_1|$. As the points of P are given in sorted order, we can obtain the above sorted list in $O(n)$ time. During the algorithm, we will compute a subset $Q \subseteq P$. Initially, we set $Q = \emptyset$. After the algorithm finishes, we will have $P' = P \setminus Q$.

We find the smallest index $i \in [1, m-1]$ such that $x(p_{i+1}) - x(p_i) > r$ (let $i = m$ if such index does not exist). It is easy to see for any point p_j with $j \in [i+1, m]$, there is no path from s to p_j in $G_r(P)$. We add all points $p_{i+1}, p_{i+2}, \ldots, p_m$ to Q and let $P'_1 = \{p_1, \ldots, p_i\}$. Hence, P'_1 has the following property: $x(p_{j+1}) - x(p_j) \leq r$ for any two adjacent points p_j and p_{j+1} (see Fig. 3.2). Next, we compute the vertical partition lines of $\Psi_r(P)$ to the right of s. We first put a vertical line through s. Then, we keep adding a vertical line to the right with horizontal distance $r/\sqrt{2}$ from the previous vertical line until the current vertical

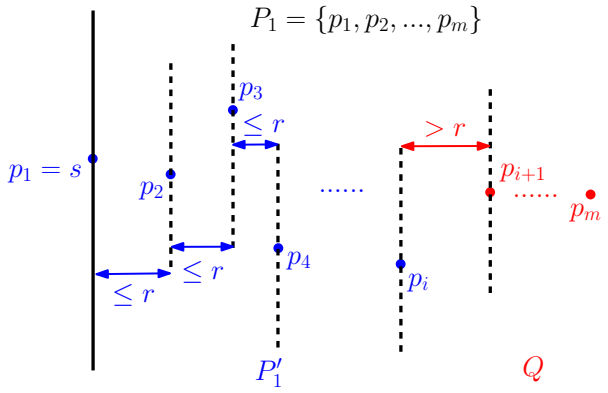


Fig. 3.2: $P_1 = \{p_1 = s, p_2, ..., p_m\}$ includes all points of P to the right of s sorted from left to right. i is the smallest index such that $x(p_{i+1}) - x(p_i) > r$. We have $P_1' = \{p_1, p_2, ..., p_i\}$. Points of $\{p_{i+1}, p_{i+2}, ..., p_m\}$ are added to the set Q since they can not be reached from s in $G_r(P)$.

line is to the right of p_i. Due to the above property of P_1', the number of vertical lines thus produced is at most $2m$.

The above computes a set of vertical partition lines to the right of s by considering the points of P_1 from left to right. Let $P_2 = P \setminus P_1$; we also add s to P_2. Symmetrically, we compute a set of vertical partition lines to the left of s by considering the points of P_2 from right to left (also starting from s). Analogously, the algorithm will compute a subset P_2' of P_2 and more points may be added to Q. Let L_v be the set of all these vertical lines produced above for both P_1 and P_2. L_v is the set of vertical partition lines of our grid $\Psi_r(P)$. Clearly, $|L_v| = O(n)$.

Similarly, by considering the points of P in the list sorted by y-coordinate, we can compute a set L_h of horizontal partition lines of $\Psi_r(P)$, with $|L_h| = O(n)$. Also, more points may be added to Q in the process.

Let $\Psi_r(P)$ be the rectangle bounded by the rightmost and leftmost vertical lines of L_v as well as the topmost and bottommost horizontal lines of L_h, along with the square cells inside and partitioned by the lines of $L_v \cup L_h$. Let $P' = P \setminus Q$. By our definition of Q, for each $p \in Q$, p cannot be reached from s in $G_r(P)$, and P' is exactly the subset of points of P located inside $\Psi_r(P)$.

For each cell C of $\Psi_r(P)$, we define its *grid-coordinate* as (i, j) if C is in the i-th row and

j-th column of $\Psi_r(P)$; we say that i is the *row-coordinate* and j is the *column-coordinate*. For each cell, we consider its grid-coordinate as its "ID".

By scanning the points of P' and the vertical lines of L_v from left to right and then scanning P' and the horizontal lines of L_h from top to bottom, we can compute in $O(n)$ time for each point of P' the (grid-coordinate of the) cell of $\Psi_r(P)$ that contains it (to resolve the boundary case, if a point p is on a vertical edge shared by two cells, then we assume p is contained in the right cell only, and if p is on a horizontal edge shared by two cells, then we assume p is contained in the top cell only). After that, given any point $p \in P'$, the cell of $\Psi_r(P)$ that contains p can be obtained in $O(1)$ time.

To compute the set $\mathcal{C}$, we do the following. Initialize $\mathcal{C} = \emptyset$. Then, for each point $p \in P'$, we add the cell that contains p into $\mathcal{C}$. Note that $\mathcal{C}$ may be a multi-set. To remove the duplicates, we first sort all cells of $\mathcal{C}$ by their grid-coordinates in lexicographical order (i.e., compare row-coordinates first and then column-coordinates). This sorting can be done in $O(n)$ time by radix sort [53], because both the row-coordinate and the column-coordinate of each cell are in the range $[1, O(n)]$. Now we can remove duplicates by simply scanning the sorted list of all cells, and the resulting set is $\mathcal{C}$. Also, during the scanning process, we can obtain for each cell C of $\mathcal{C}$ the subset $P(C)$ of points of P contained in C (each occurrence of C in the sorted list corresponds to a point of P that is contained in C). All these can be done in $O(n)$ time. After that, given each cell C of $\mathcal{C}$, we can output $P(C)$ in $O(|P(C)|)$ time.

It remains to compute the neighbor set $N(C)$ for each cell $C \in \mathcal{C}$. This can be done in $O(n)$ time by scanning the above sorted list of $\mathcal{C}$ (after the duplicates are removed). Indeed, notice that scanning the sorted list is equivalent to scanning the non-empty cells of $\Psi_r(P)$ row by row and from left to right in each row. Recall that the cells of $N(C)$ are in at most five rows of the grid (e.g., see Fig. 3.1): the row containing C, two rows above it, and two rows below it; each such row contains at most fives cells of $N(C)$. Based on this observation, we scan the cells in the sorted list of $\mathcal{C}$. For each cell C under consideration during the scan, suppose its grid-coordinate is (i, j). During the scan, we maintain a cell

$(i', j') \in \mathcal{C}$ in each row i' for $i' \in \{i-2, i-1, i, i+1, i+2\}$ such that j' is closest to j, i.e., $|j'-j|$ is minimized (e.g., for $i' = i$, we have $j' = j$). Using these cells, we can find $N(C)$ in $O(1)$ time (indeed, for each row $i' \in \{i-2, i-1, i, i+1, i+2\}$, the cells of $N(C)$ contained in row i' are within five cells of (i', j') in the sorted list of $\mathcal{C}$). The scan can be implemented in $O(n)$ time. After that, $N(C)$ for all cells $C \in \mathcal{C}$ are computed. This proves the lemma. □

To make the description concise, in the following, whenever we say "compute the grid $\Psi_r(P)$" we mean "compute the grid information of Lemma 3.1"; similarly, by "using the grid $\Psi_r(P)$", we mean "using the grid information computed by Lemma 3.1".

3.3 The unweighted case – the first algorithm

In this section, we present our $O(\lambda \cdot n \log n)$ time algorithm for the unweighted RSP problem. Given λ and $s, t \in P$, our goal is to compute r^*, the optimal radius of the disks.

As discussed in Section 3.1.2, our algorithm uses parametric search [51, 52]. But different than the traditional parametric search where parallel algorithms are used, our decision algorithm (i.e., the CS algorithm for the shortest path problem [13]) is inherently sequential. We will run the CS algorithm with a parameter r in an interval $(r_1, r_2]$ by simulating the algorithm on the unknown r^*; at each step of the algorithm, the decision algorithm will be invoked on certain *critical values* r to compare r and r^*, and the algorithm will proceed accordingly based on the results of the comparisons. The interval $(r_1, r_2]$ always contains r^* and will keep shrinking during the algorithm (note that "shrinking" includes the case that the interval does not change). Initially, we set $r_1 = 0$ and $r_2 = \infty$. Clearly, $(r_1, r_2]$ contains r^*.

Recall that the CS algorithm has two major steps: build the grid and then run BFS with the help of the grid. Correspondingly, our algorithm also first builds a grid and then runs BFS accordingly using the grid.

3.3.1 Building the grid

The first step is to build a grid $\Psi(P)$. Our goal is to shrink $(r_1, r_2]$ so that it contains r^* and if $r^* \neq r_2$ (and thus $r^* \in (r_1, r_2)$), then for any $r \in (r_1, r_2)$, $\Psi_r(P)$ has the same *combinatorial structure* as $\Psi_{r^*}(P)$, i.e., both grids have the same number of columns and the same number of rows, and a point of P is in the cell of the i-th row and j-th column of $\Psi_{r^*}(P)$ if and only if it is also in the cell of the i-th row and j-th column of $\Psi_r(P)$. To this end, we have the following lemma.

Lemma 3.2. *An interval $(r_1, r_2]$ containing r^* can be computed in $O(n \log n)$ time so that if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, the grid $\Psi_r(P)$ has the same combinatorial structure as $\Psi_{r^*}(P)$.*

Proof. Let P_1 be the subset of P to the right of s including s. Let $s = p_1, p_2, \ldots, p_m$ be the list of P_1 sorted from left to right, with $m = |P_1|$. Recall from the proof of Lemma 3.1 that $\Psi_{r^*}(P)$ has at most $2m$ vertical partition lines to the right of s, and there is a vertical partition line through s.

We first implicitly form a sorted matrix and then apply the sorted-matrix searching techniques of Frederickson and Johnson [54–56] (specifically, see Theorem 2.1 in [54]) to shrink $(r_1, r_2]$. Specifically, we define an $m \times 2m$ matrix M with

$$M[i, j] = \sqrt{2} \cdot \frac{x(p_i) - x(p_1)}{j}$$

for all $1 \leq i \leq m$ and $1 \leq j \leq 2m$. It can be verified that $M[i, j] \geq M[i, j+1]$ and $M[i+1, j] \geq M[i, j]$ hold. Thus, M is a sorted matrix. Using the sorted-matrix searching techniques [54–56] with the CS algorithm as the decision algorithm, we can compute in $O(n \log n)$ time the largest value r_1' of M with $r_1' < r^*$ and the smallest value r_2' of M with $r^* \leq r_2'$. By definition, $(r_1', r_2']$ contains r^* and (r_1', r_2') does not contain any value of M. We update $r_1 = \max\{r_1', r_1\}$ and $r_2 = \min\{r_2', r_2\}$. Thus, the new interval $(r_1, r_2]$ shrinks but still contains r^*. As $(r_1, r_2) \subseteq (r_1', r_2')$, (r_1, r_2) does not contain any value of M.

According to our algorithm of Lemma 3.1, there is always a vertical partition line through s in $\Psi_r(P)$ for any r. Let $\Psi_r^1(P)$ and $\Psi_r^2(P)$ refer to the half grids of $\Psi_r(P)$ to the

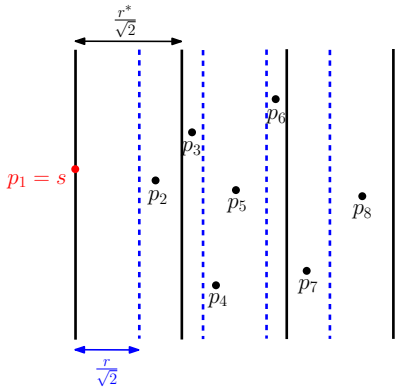


Fig. 3.3: The point p_7 (the red point) is in the 3rd column of $\Psi^1_{r^*}(P)$ while it is in the 4th column of $\Psi^1_r(P)$.

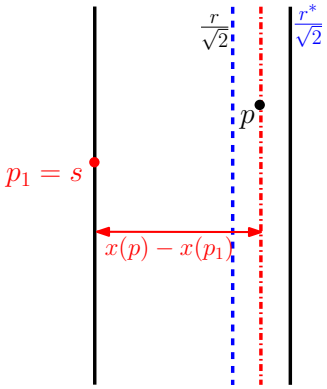


Fig. 3.4: The rightmost line is ℓ when $r' = r^*$. When r' decreases from r^* to r, ℓ will move leftwards and cross p.

right and left of s, respectively; assume that both half grids contain the vertical partition line through s. We claim that if $r^* \neq r_2$, then the following hold for any $r \in (r_1, r_2)$: (1) a point of P_1 is in the j-th column of $\Psi^1_{r^*}(P)$ if and only if it is also in the j-th column of Ψ^1_r(P); (2) the number of columns of $\Psi^1_{r^*}(P)$ is equal to the number of columns of $\Psi^1_r(P)$. We prove the claim below.

Suppose $r^* \neq r_2$. Then, $r^* \in (r_1, r_2)$. Assume to the contrary that a point p of P_1 is in the j-th column of $\Psi^1_{r^*}(P)$ for some $j \in [1, 2m]$, but p is not in the j-th column of $\Psi^1_r(P)$. Then, p is either to the left or to the right of the j-th column of $\Psi^1_r(P)$. Without loss of generality, we assume that p is to the right of the j-th column of $\Psi^1_r(P)$ (e.g., see Fig. 3.3). This implies that $r < r^*$. Further, if we decrease a value r' gradually from r^* to r, then the line ℓ will move monotonically leftwards and cross p at some moment, where ℓ is the $(j+1)$-th vertical partition line of $\Psi^1_{r'}(P)$ (i.e., ℓ is the vertical bounding line of the j-th column of $\Psi^1_{r'}(P)$); e.g., see Fig. 3.4. This further implies that $r/\sqrt{2} < (x(p) - x(p_1))/j < r^*/\sqrt{2}$, and thus, $r < \sqrt{2} \cdot (x(p) - x(p_1))/j < r^*$. On the other hand, since both r and r^* are in (r_1, r_2), we obtain that $\sqrt{2} \cdot (x(p) - x(p_1))/j \in (r_1, r_2)$. Because the interval (r_1, r_2) does not contain any values of M, we obtain a contradiction as $\sqrt{2} \cdot (x(p) - x(p_1))/j$ is a value of M.

Assume to the contrary that a point p of P_1 is in the j-th column of $\Psi^1_r(P)$ for some $j \in [1, 2m]$, but p is not in the j-th column of $\Psi^1_{r^*}(P)$. Then, by a similar analysis as above,

we can obtain a contradiction as well. This proves the first part of the claim.

The second part of the claim can actually be derived by the first part. Indeed, assume to the contrary that the number of columns of $\Psi^1_{r^*}(P)$, denoted by m_{r^*}, is not equal to the number of columns of $\Psi^1_r(P)$, denoted by m_r. Without loss of generality, we assume $m_{r^*} < m_r$. By the algorithm of Lemma 3.1, P_1 has a point p in the last column of $\Psi^1_r(P)$, which is the m_r-th column. In light of the first part of the claim, p is also in the m_r-th column of $\Psi^1_{r^*}(P)$. But this contradicts with that $\Psi^1_{r^*}(P)$ has only $m_{r^*} < m_r$ columns.

The claim is thus proved.

The above processes the subset P_1 of P. Let $P_2 = P \setminus P_1$; we add s to P_2 as well. Next, we use the same algorithm as above to process the points of P_2 and obtain a smaller interval $(r_1, r_2]$ containing r^* such that if $r^* \neq r_2$, then the following hold for any $r \in (r_1, r_2)$: (1) a point of P_2 is in the j-th column of $\Psi^2_{r^*}(P)$ if and only if it is also in the j-th column of Ψ^2_r(P); (2) the number of columns of $\Psi^2_{r^*}(P)$ is equal to the number of columns of $\Psi^2_r(P)$. Combining the previous claim for P_1, we obtain that the interval $(r_1, r_2]$ contains r^* and if $r^* \neq r_2$, then the following hold for any $r \in (r_1, r_2)$: (1) a point of P is in the j-th column of $\Psi_{r^*}(P)$ if and only if it is also in the j-th column of $\Psi_r(P)$; (2) the number of columns of $\Psi_{r^*}(P)$ is equal to the number of columns of $\Psi_r(P)$.

The above processes the points of P horizontally. We then process them in a vertical manner analogously and further shrink the interval $(r_1, r_2]$ such that it still contains r^* and if $r^* \neq r_2$, then the following hold for any $r \in (r_1, r_2)$: (1) a point of P is in the i-th row of $\Psi_{r^*}(P)$ if and only if it is also in the i-th row of $\Psi_r(P)$; (2) the number of rows of $\Psi_{r^*}(P)$ is equal to the number of rows of $\Psi_r(P)$. As the interval $(r_1, r_2]$ is shrunk after processing P vertically, we obtain that if $r^* \neq r_2$, then $\Psi_r(P)$ has the same combinatorial structure as $\Psi_{r^*}(P)$ for any $r \in (r_1, r_2)$. This proves the lemma. □

Let $(r_1, r_2]$ be the interval computed by Lemma 3.2. We pick any value r in (r_1, r_2) and compute the grid $\Psi_r(P)$, i.e., compute the grid information of $\Psi_r(P)$ by Lemma 3.1. By Lemma 3.2, these information is the same as that of $\Psi_{r^*}(P)$ if $r^* \neq r_2$. Below we will use $\Psi(P)$ to refer to the grid information computed above.

3.3.2 Running BFS

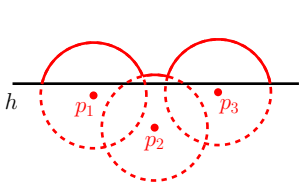
For a fixed parameter r, we use $S_i(r)$ to denote the set of points of P whose distances from s is equal to i in $G_r(P)$, which is computed in the i-th step of the BFS algorithm if we run the CS algorithm with respect to r. Initially, we have $S_0(r) = \{s\}$. In the following, using the interval $(r_1, r_2]$ obtained in Lemma 3.2, we run the BFS algorithm as in the CS algorithm with a parameter $r \in (r_1, r_2)$, by simulating the algorithm for r^*. The algorithm maintains an invariant that the i-th step computes a subset $S_i \subseteq P$ and shrinks $(r_1, r_2]$ so that it contains r^* and if $r^* \neq r_2$ (and thus $r^* \in (r_1, r_2)$), then $S_i = S_i(r) = S_i(r^*)$ for any $r \in (r_1, r_2)$. Initially, we set $S_0 = \{s\}$ and thus the invariant holds as $S_0(r) = \{s\}$ for any r. As will be seen later, the algorithm stops within λ steps and each step takes $O(n \log n)$ time.

Consider the i-th step. Assume that we have S_{i-1} and $(r_1, r_2]$, and the invariant holds, i.e., $(r_1, r_2]$ contains r^* and if $r^* \neq r_2$, then $S_{i-1} = S_{i-1}(r) = S_{i-1}(r^*)$ for any $r \in (r_1, r_2)$. Using the grid $\Psi(P)$, we obtain the grid cells containing the points of S_{i-1}. For each such cell C, for points of P in C, we have the following observation.

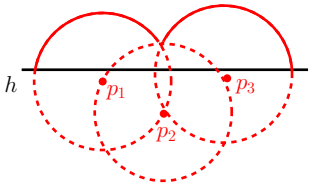
Lemma 3.3. *Suppose $r^* \neq r_2$. Then, for each point $p \in P(C)$ that has not been discovered by the algorithm yet, i.e., $p \notin \bigcup_{j=1}^{i-1} S_j$, p is in $S_i(r)$ for all $r \in (r_1, r_2)$.*

Proof. Let q be a point of S_{i-1} in C. By our algorithm invariant, $(r_1, r_2]$ contains r^*. Since $r^* \neq r_2$, $r^* \in (r_1, r_2)$. Let r be any value of (r_1, r_2). In light of Lemma 3.2, both p and q are in the same cell of $\Psi_r(P)$, and thus $\|p - q\| \leq r$. By our algorithm invariant, $S_j = S_j(r)$ for all $0 \leq j \leq i-1$. Since $p \notin \bigcup_{j=1}^{i-1} S_j$, we have $p \notin \bigcup_{j=1}^{i-1} S_j(r)$. Because $q \in S_{i-1}(r)$ and $\|p - q\| \leq r$, we obtain that $p \in S_i(r)$. □

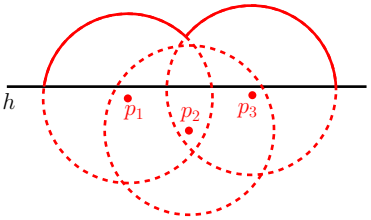
Due to the preceding lemma, we add to S_i the points of $P(C)$ that have not been discovered yet. Next, for each neighbor C' of C, we need to solve Subproblem 3.1; we use $\mathcal{I}$ to denote the set of all instances of this subproblem in the i-th step of the BFS. Consider one such instance. Recall that solving it for a fixed r involves three subroutines. First, compute the upper envelope $\mathcal{U}$ of the arcs of Γ above ℓ of all red points. Second, sort all vertices of $\mathcal{U}$ with all blue points. Third, for each blue point p, determine whether it is



(a) The upper envelope is comprised of three arcs centered at p_1, p_2 and p_3.



(b) The moment when the three arcs have a common intersection, which is a vertex of the upper envelope.



(c) The middle arc centered at p_2 disappears from the upper envelope.

Fig. 3.5: The change of the combinatorial structure of the upper envelope $\mathcal{U}(r)$ (the red solid arcs) as r increases.

below the arc of $\mathcal{U}$ that spans p. To solve our problem, we parameterize each subroutine with a parameter r so that the behavior of the algorithm is consistent with that for $r = r^*$ if $r^* \neq r_2$.

Computing the upper envelope

We use $\Gamma(r)$ to denote the set of arcs above ℓ defined by the red points with respect to the radius r; similarly, define $\mathcal{U}(r)$ as the upper envelope of $\Gamma(r)$.

The goal of the first subroutine is to shrink the interval $(r_1, r_2]$ such that it contains r^* and if $r^* \neq r_2$, then $\mathcal{U}(r^*)$ has the same combinatorial structure as $\mathcal{U}(r)$ for any $r \in (r_1, r_2)$, i.e., the set of red points that define the arcs on $\mathcal{U}(r)$ is exactly the set of red points that define the arcs on $\mathcal{U}(r^*)$ with the same order. Note that the order of the arcs on $\mathcal{U}(r)$ is consistent with the x-coordinate order of the red points defining these arcs [13].

To this end, we have the following observation. Consider $\mathcal{U}(r)$ for an arbitrary r. If r changes, the combinatorial structure of $\mathcal{U}(r)$ does not change until one arc (e.g., defined by a red point p_2) disappears from $\mathcal{U}(r)$ (e.g., see Fig. 3.5). Let p_1 and p_3 be the red points

defining neighboring left and right arcs of the arc defined by p_2 on $\mathcal{U}(r)$, respectively. Then, at the moment when p_2 disappears from $\mathcal{U}(r)$, the three arcs defined by p_1, p_2, and p_3 intersect at a common point q, which is equidistant to the three points. Further, since q is currently on $\mathcal{U}(r)$, there is no red point that is closer to q than p_i for $i = 1, 2, 3$, and the distance from q to each p_i, $i = 1, 2, 3$, is equal to the current value of r. Hence, q is a vertex of the Voronoi diagram of the red points. This implies that as r changes, the combinatorial structure of $\mathcal{U}(r)$ does not change until possibly when r is equal to the distance $\|q - p\|$, where q is a vertex of the Voronoi diagram of all red points and p is a nearest red point of q.

Based on the above observation, our algorithm works as follows. We build the Voronoi diagram for all red points, which takes $O(n_r \log n_r)$ time [57, 58]. For each vertex v of the diagram, we add $\|v - p\|$ to the set $\mathcal{Q}$ (initially $\mathcal{Q} = \emptyset$), where p is a nearest red point of v (p is available from the diagram). Note that $|\mathcal{Q}| = O(n_r)$, and we refer to each value of $\mathcal{Q}$ as a *critical value*. Next, we sort $\mathcal{Q}$, and then do binary search on $\mathcal{Q}$ using the decision algorithm to find the smallest value r'_2 of $\mathcal{Q}$ with $r'_2 \geq r^*$ as well as the largest value r'_1 of $\mathcal{Q}$ smaller than r^*, which can be done in $O(n \log n_r)$ time (note that $n_r \leq n$). By definition, $(r'_1, r'_2]$ contains r^* and (r'_1, r'_2) does not contain any value of $\mathcal{Q}$. According to the above observation, if $r^* \neq r'_2$, then the combinatorial structure of $\mathcal{U}(r^*)$ is the same as that of $\mathcal{U}(r)$ for any $r \in (r'_1, r'_2)$.

We analyze the running time of this subroutine for all instances of $\mathcal{I}$. Clearly, the total time for all instances is bounded by $O(|\mathcal{I}| \cdot n \log n)$, which is $O(n^2 \log n)$ as $|\mathcal{I}| = O(n)$. We can reduce the time to $O(n \log n)$ by considering the critical values of all instances of $\mathcal{I}$ all together. Specifically, let $\mathcal{Q}$ now be the set of critical values of all instances of $\mathcal{I}$. Then, $|\mathcal{Q}| = O(n)$. We sort $\mathcal{Q}$ and do binary search on $\mathcal{Q}$ to find r'_1 and r'_2 as defined above with respect to the new $\mathcal{Q}$. Now, for each instance of $\mathcal{I}$, if $r^* \neq r'_2$, then the combinatorial structure of $\mathcal{U}(r^*)$ is the same as that of $\mathcal{U}(r)$ for any $r \in (r'_1, r'_2)$. The total time for all instances of $\mathcal{I}$ is now bounded by $O(n \log n)$. Finally, we update $r_1 = \max\{r_1, r'_1\}$ and $r_2 = \min\{r_2, r'_2\}$. As $r^* \in (r'_1, r'_2]$, the new interval $(r_1, r_2]$ still contains r^*. Further, as

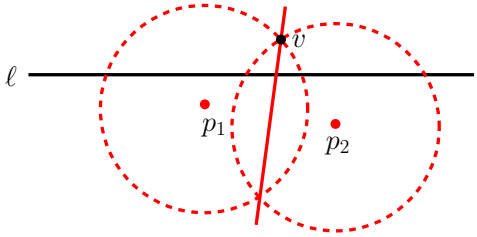


Fig. 3.6: Illustrating a vertex v of the upper envelope, which is defined by two red points p_1 and p_2. The red solid segment is the bisector of p_1 and p_2.

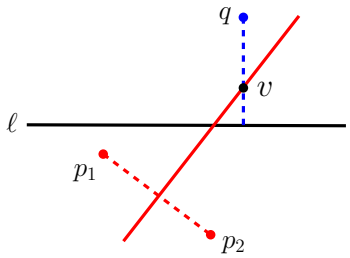


Fig. 3.7: Illustrating the scenario where $x(q) = x(v)$, where v is on the bisector (the red solid segment) of p_1 and p_2.

$(r_1, r_2) \subseteq (r_1', r_2')$, for each instance of $\mathcal{I}$, if $r^* \neq r_2$, then the combinatorial structure of $\mathcal{U}(r^*)$ is the same as that of $\mathcal{U}(r)$ for any $r \in (r_1, r_2)$.

Sorting the upper envelope vertices and blue points

The goal of the second subroutine is to shrink the interval $(r_1, r_2]$ such that it contains r^* and if $r^* \neq r_2$, then the sorted list of all vertices of $\mathcal{U}(r^*)$ and all blue points by their x-coordinates is the same as the sorted list of all vertices of $\mathcal{U}(r)$ and all blue points for any $r \in (r_1, r_2)$.

Recall that after the first subroutine, the interval $(r_1, r_2]$ contains r^*, and if $r^* \neq r_2$, then the combinatorial structure of $\mathcal{U}(r^*)$ is the same as that of $\mathcal{U}(r)$ for any $r \in (r_1, r_2)$.

To sort all vertices of $\mathcal{U}(r^*)$ and all blue points, we apply Cole's parametric search [51] with AKS sorting network [59], using the CS algorithm as the decision algorithm; the running time is bounded by $O(n \log n)$ as the number of vertices of $\mathcal{U}(r^*)$ is $O(n_r)$ and the number of blue points is $O(n_b)$ (and $n_r + n_b = O(n)$). To see why this works, it suffices to argue that the "root" of each comparison involved in the sorting can be obtained in $O(1)$ time (more specifically, the root refers to the value of $r \in (r_1, r_2)$ at which the two operands involved in the comparison are equal). Indeed, the comparisons can be divided into three types based on their operands: (1) a comparison between the x-coordinates of two blue points; (2) a comparison between the x-coordinates of two vertices of $\mathcal{U}(r^*)$; (3) a comparison between the x-coordinates of a blue point and a vertex of $\mathcal{U}(r^*)$. For the first type, as blue points are fixed, independent of the parameter r, it is trivial to handle. For the

second type, as the combinatorial structure of $\mathcal{U}(r)$ does not change for all $r \in (r_1, r_2)$, each such comparison can be resolved by taking any value of $r \in (r_1, r_2)$ and then comparing the two vertices under r. The third type is a little more involved. Consider the comparison of the x-coordinates of a blue point q and a vertex v of $\mathcal{U}(r^*)$. Note that v is the intersection of arcs of two circles of radius r and centered at two red points, say p_1 and p_2, respectively. Observe that v is on the bisector of p_1 and p_2 (e.g., see Fig. 3.6). Furthermore, when r changes, v moves on the bisector of p_1 and p_2, while the position of the blue point q does not change. Hence, the root of the comparison, i.e., the value r (if exists) in (r_1, r_2) such that $x(q) = x(v)$ can be obtained in constant time by elementary geometry (e.g., see Fig. 3.7). Note that if such r does not exist in (r_1, r_2), then either $x(q) < x(v)$ holds for all $r \in (r_1, r_2)$ or $x(q) > x(v)$ holds for all $r \in (r_1, r_2)$, which can be easily determined. As such, with Cole's parametric search [51] and the linear time decision algorithm (i.e., the CS algorithm), we can obtain a sorted list of the upper envelope vertices and the blue points by their x-coordinates; the algorithm shrinks the interval $(r_1, r_2]$ so that the new interval $(r_1, r_2]$ contains r^* and if $r^* \neq r_2$, then the above sorted list is fixed for all $r \in (r_1, r_2)$.

Since the running time of the above sorting algorithm is $O(n \log n)$, as before for the first subroutine, the sorting for all problem instances of $\mathcal{I}$ takes $O(n^2 \log n)$ time. To reduce the time, as before, we sort all elements in all instances of $\mathcal{I}$ altogether, which takes $O(n \log n)$ time in total. Specifically, in each problem instance, we need to sort a set of blue points and vertices of upper envelopes of a set of red points. We put all blue points and the upper envelopes of all red points of all problem instances of $\mathcal{I}$ in one coordinate system and apply the sorting algorithm as above. One difference is that we now have a new type of comparisons: compare the x-coordinate of a vertex v_1 of the upper envelope from one problem instance with the x-coordinate of a vertex v_2 of the upper envelope from another problem instance. In this case, when r changes, both v_1 and v_2 moves on the bisectors of their defining red points. But we can still find in constant time a root r (if exists) in (r_1, r_2) for the comparison by elementary geometry. As such, we can complete the sorting for all problem instances of $\mathcal{I}$ in $O(n \log n)$ time in total, for the total number of all blue points

and red points in all problem instances of $\mathcal{I}$ is $O(n)$. Again, the interval $(r_1, r_2]$ will be shrunk. This finishes the second subroutine.

Deciding whether each blue point is below the upper envelope

We now have an interval $(r_1, r_2]$ containing r^* such that if $r^* \neq r_2$, then each blue point q is spanned by an arc $\alpha_q(r)$ of $\mathcal{U}(r)$ defined by the same red point for all $r \in (r_1, r_2)$ (note that the arc $\alpha_q(r)$ moves as r changes, for r is the radius of the arc). Each blue point q is below the upper envelope $\mathcal{U}(r)$ if and only if q is below the arc $\alpha_q(r)$. The goal of the third subroutine is to shrink the interval $(r_1, r_2]$ so that the new interval $(r_1, r_2]$ still contains r^* and if $r^* \neq r_2$, then for each blue point q, the relative position of q with respect to $\alpha_q(r)$ (i.e., whether q is above or below $\alpha_q(r)$) is fixed for all $r \in (r_1, r_2)$. To this end, we proceed as follows.

As r changes in (r_1, r_2), $\alpha_q(r)$ changes while q does not. For each blue point q, we compute in constant time a critical value r (if exists) in (r_1, r_2) such that q is on α_q, and we add r to the set $\mathcal{Q}$ ($\mathcal{Q} = \emptyset$ initially). Note that if such value r does not exist in (r_1, r_2), then either q is above $\alpha_q(r)$ for all $r \in (r_1, r_2)$ or q is below $\alpha_q(r)$ for all $r \in (r_1, r_2)$, which can be easily determined. The size of $\mathcal{Q}$ is at most n_b. Then, we sort $\mathcal{Q}$, and do binary search on $\mathcal{Q}$ with our decision algorithm to find the smallest value r_2' of $\mathcal{Q}$ with $r_2' \geq r^*$ and the largest value r_1' of $\mathcal{Q}$ with $r_1' < r^*$. We then update $r_1 = \max\{r_1, r_1'\}$ and $r_2 = \min\{r_2, r_2'\}$. The new interval $(r_1, r_2]$ still contains r^* and (r_1, r_2) does not contain any value of $\mathcal{Q}$. Hence, if $r^* \neq r_2$, then for each blue point q, the relative position of q with respect to $\alpha_q(r)$ is fixed for all $r \in (r_1, r_2)$. As such, the new interval $(r_1, r_2]$ satisfies the goal of the third subroutine as mentioned above.

Finally, we pick an arbitrary $r \in (r_1, r_2)$, and for each blue point q, if q is below the arc $\alpha_q(r)$, then we add q to the set S_i.

The running time of the above algorithm is $O(n \log n_b)$. Thus the total time of the third subroutine is $O(n^2 \log n)$ for all problem instances of $\mathcal{I}$. To reduce the time, we again consider the subroutine of all instances of $\mathcal{I}$ altogether. More specifically, we put all critical values r in all problem instances of $\mathcal{I}$ in $\mathcal{Q}$. Thus, the size of $\mathcal{Q}$ is $O(n)$. We then run the

same algorithm as above using the new set $\mathcal{Q}$. The total time is bounded by $O(n \log n)$.

Terminating the algorithm

This finishes the i-th step of the BFS, which computes a set S_i along with an interval $(r_1, r_2]$. According to the above discussion, $(r_1, r_2]$ contains r^* and if $r^* \neq r_2$ (and thus $r^* \in (r_1, r_2)$), then $S_i = S_i(r^*) = S_i(r)$ for all $r \in (r_1, r_2)$.

If the point t is in S_i and $i \leq \lambda$, then we stop the algorithm. In this case, we have the following lemma.

Lemma 3.4. *If $t \in S_i$ and $i \leq \lambda$, then $r^* = r_2$.*

Proof. Assume to the contrary that $r^* \neq r_2$. Then, since $r^* \in (r_1, r_2]$, we have $r^* \in (r_1, r_2)$. Let $r' = (r_1 + r^*)/2$. Clearly, $r' \in (r_1, r_2)$ and $r' < r^*$. As $r' \in (r_1, r_2)$, $S_i = S_i(r')$ by our algorithm invariant. Since $t \in S_i(r')$, we obtain that $d_{r'}(s, t) = i \leq \lambda$. This leads to a contradiction as $r' < r^*$ and r^* is the minimum value r with $d_r(s, t) \leq \lambda$. □

If $t \notin S_i$ and $i = \lambda$, then we also stop the algorithm. In this case, we have the following lemma.

Lemma 3.5. *If $t \notin S_i$ and $i = \lambda$, then $r^* = r_2$.*

Proof. Assume to the contrary that $r^* \neq r_2$. Then, $r^* \in (r_1, r_2)$, for $r^* \in (r_1, r_2]$. By our algorithm invariant, $S_j = S_j(r)$ for all $r \in (r_1, r_2)$ and for all $j \leq i$. Hence, $S_j = S_j(r^*)$ for all $j \leq i$. As $t \notin S_i$, according to our algorithm, $t \notin \bigcup_{j=0}^{i} S_j$. Therefore, $t \notin \bigcup_{j=0}^{i} S_j(r^*)$, implying that $d_{r^*}(s, t) > i = \lambda$. However, by the definition of r^*, $d_{r^*}(s, t) \leq \lambda$ holds. We thus obtain contradiction. □

Since initially $i = 0$ and $S_0 = \{s\}$, the above implies that the BFS algorithm will stop in at most λ steps. As each step takes $O(n \log n)$ time, the value r^* can be computed in $O(\lambda \cdot n \log n)$ time.

Theorem 3.1. *The reverse shortest path problem for L_2 unweighted unit-disk graphs can be solved in $O(\lfloor \lambda \rfloor \cdot n \log n)$ time.*

3.4 The unweighted case – the second algorithm

In this section, we present our second algorithm for the L_2 unweighted RSP problem. As discussed in Section 3.1.2, the main idea is to combine the strategies of the first unweighted RSP algorithm in Section 3.3 and the naive binary search algorithm using the distance selection algorithm [30].

First of all, we still build in $O(n \log n)$ time the grid $\Psi(P)$ as in Section 3.3.1, and thus the information of Lemma 3.2 is available for the grid. More specifically, we obtain an interval $(r_1, r_2]$ such that if $r^* \neq r_2$, then the combinatorial data structure of $\Psi_r(P)$ is fixed for all $r \in (r_1, r_2)$, implying that $\mathcal{C}$, P', $N(C)$ and $P(C)$ for each $C \in \mathcal{C}$ are fixed for all $r \in (r_1, r_2)$. Next, we will run the BFS algorithm, but in a different way than before.

We partition the cells of $\mathcal{C}$ into *large cells* and *small cells*: a cell C is a large cell if $|P(C)| \geq (n/\log n)^{3/4}$ and is a small cell otherwise. Thus the number of large cells is at most $n^{1/4} \log^{3/4} n$. For all pairs of cells (C, C') with $C \in \mathcal{C}$ and $C' \in N(C)$, we call (C, C') a *small-cell pair* if both C and C' are small cells and a *large-cell pair* otherwise (i.e., at least one cell is a large cell). As $|N(C)| = O(1)$ for each cell C and the number of large cells is at most $n^{1/4} \log^{3/4} n$, the total number of large-cell pairs is $O(n^{1/4} \log^{3/4} n)$.

Recall that each step of the BFS algorithm of our first algorithm in Section 3.3.2 boils down to solving instances of Subproblem 3.1, and each such instance involves a cell pair (C, C') with $C \in \mathcal{C}$ and $C' \in N(C)$. If (C, C') is a large-cell pair, we will run the same algorithm as in Section 3.3.2. Otherwise, we will use the original CS algorithm to solve it, which takes only linear time. For this, with the help of the L_2 distance selection algorithm [30], we preprocess all these small-cell pairs before starting the BFS algorithm by the following lemma.

Lemma 3.6. *An interval $(r'_1, r'_2]$ containing r^* can be computed in $O(n^{5/4} \log^{7/4} n)$ time with the following property: if $r^* \neq r'_2$, then for any $r \in (r'_1, r'_2)$, for any small-cell pair (C, C') with $C \in \mathcal{C}$ and $C' \in N(C)$, an edge connects a point $p \in P(C)$ and a point $p' \in P(C')$ in $G_r(P)$ if and only if an edge connects p and p' in $G_{r^*}(P)$.*

Proof. Let Π denote the set of all small-cell pairs (C, C') with $C \in \mathcal{C}$ and $C' \in N(C)$. We use (C_i, C_i') to denote the i-th pair of Π; let P_i denote the set of points of P in the two cells C_i and C_i', and let $n_i = |P_i|$. Let $m = |\Pi|$. Note that $m = O(n)$. By the definition of small cells, we have $n_i \leq 2 \cdot (n/\log n)^{3/4}$. Since $|N(C)| = O(1)$ for each cell C, it holds that $\sum_{i=1}^{m} n_i = O(n)$. For each P_i, let D_i denote the set of distances of all pairs of points of P_i. Hence, $|D_i| = n_i(n_i - 1)/2$. Define $\mathcal{D} = \bigcup_{i=1}^{m} D_i$.

Let r_2' be the smallest value of $\mathcal{D}$ with $r_2' \geq r^*$ and let r_1' be the largest value of $\mathcal{D}$ smaller than r^*. By definition, $(r_1', r_2']$ contains r^* and the open interval (r_1', r_2') does not contain any value of $\mathcal{D}$ and thus any value of D_i for each i. Therefore, for any two points p and p' of P_i, either $\|p - p'\| < r$ holds for all $r \in (r_1', r_2')$ or $\|p - p'\| > r$ holds for all $r \in (r_1', r_2')$. Thus, $(r_1', r_2']$ satisfies the lemma statement. In the following, we only describe the algorithm for finding r_2' since the algorithm for finding r_1' is similar.

For convenience, for any r, we say that r is *feasible* if $r \geq r^*$ and *infeasible* otherwise. Note that if r is a feasible value, then r' is also feasible for any $r' > r$; symmetrically, if r is infeasible, then r' is also infeasible for any $r' < r$. Recall that given any r, we can decide whether $r \geq r^*$ in linear time using the decision algorithm (i.e., the CS algorithm).

For each P_i, we wish to do binary search on all distances of D_i. However, doing this on each P_i individually would be time-consuming. Instead, we do binary search for all P_i's all together in a "batched" way. Specifically, for each P_i, we use the L_2 distance selection algorithm [30] to compute the median distance of D_i, denoted by d_i, which takes $O(n_i^{4/3} \log^2 n_i)$ time. Then, we sort all these medians d_i's, for all $i = 1, 2, \ldots, m$, and do binary search on the sorted list using the decision algorithm. In $O(n \log n)$ time, we can determine whether each d_i is feasible. Among all these medians, we keep the smallest feasible value, denoted by d^1. This finishes the first round of the algorithm.

In the second round, for each d_i, if it is feasible, then any value of D_i larger than d_i is also feasible; in this case, we compute the $(|D_i|/4)$-th smallest value of D_i, denoted by d_i'. If d_i is infeasible, then any value of D_i smaller than d_i is also infeasible; in this case, we compute the $(3|D_i|/4)$-th smallest value of D_i, denoted by d_i'. Next, we determine whether

the values d'_i are feasible for all $1 \leq i \leq m$ in the same way as above (i.e., doing binary search using the decision algorithm); we keep the smallest feasible value, denoted by d^2.

We then continue the next round in a similar way as above. After $O(\log n)$ rounds, the values of all sets D_i are processed and we obtain a set of $O(\log n)$ feasible values d^1, d^2, ...; among all these values, the smallest one is r'_2.

For the time analysis, the algorithm has $O(\log n)$ rounds and each round takes $O(n \log n + \sum_{i=1}^{m} n_i^{4/3} \log^2 n_i)$ time. Since $n_i \leq 2 \cdot (n/\log n)^{3/4}$ for each $1 \leq i \leq m$, and $\sum_{i=1}^{m} n_i = O(n)$, the sum $\sum_{i=1}^{m} n_i^{4/3}$ achieves maximum when each n_i is equal to $2 \cdot (n/\log n)^{3/4}$ (and thus $m = O(n^{1/4} \log^{3/4} n)$). Hence, $\sum_{i=1}^{m} n_i^{4/3} = O(n^{5/4}/\log^{1/4} n)$. Therefore, each round of the algorithm takes $O(n^{5/4} \log^{7/4} n)$ time, which is dominated by the L_2 distance selection algorithm [30]. The total time of the algorithm is thus $O(n^{5/4} \log^{11/4} n)$.

In what follows, we reduce the runtime of the algorithm by a logarithmic factor. The new algorithm still has $O(\log n)$ rounds. The difference is that instead of applying the L_2 distance selection algorithm [30] directly, we only use a subroutine of that algorithm. This also simplifies the overall algorithm. To avoid the lengthy background discussion, we use concepts from [30] without further explanation (refer to the initial version of the algorithm in Section 4 [30] for the details).

Each round of our algorithm produces an interval $I_j = (a_j, b_j]$ which contains r^*. Initially, we set $I_0 = (0, \infty]$; we also add ∞ to $\mathcal{D}$. Given an interval $I_{j-1} = (a_{j-1}, b_{j-1}]$ that contains r^* with $b_{j-1} \in \mathcal{D}$, the j-th round of the algorithm produces an interval $I_j = (a_j, b_j]$ that also contains r^* with $b_j \in \mathcal{D}$ such that $I_j \subseteq I_{j-1}$ and the number of values of $\mathcal{D}$ contained in I_j is only a constant fraction of the number of values of $\mathcal{D}$ contained in I_{j-1}. Thus, after $O(\log n)$ rounds, we are left with a sufficiently small number of distances of $\mathcal{D}$, from which it is trivial to find r'_2.

The j-th round of the algorithm works as follows. For each set P_i, we compute a compact representation of all pairs of points of P_i whose distances lie in I_{i-1}, which can be done in $O(n_i^{4/3} \log n_i)$ time [30]. Such a compact representation is a collection of $O(n_i^{4/3})$ complete bipartite graphs $\{Q_k \times W_k\}_k$, where both $\sum_k |Q_k|$ and $\sum_k |W_k|$ are bounded by

$O(n_i^{4/3} \log n_i)$. For each k, the distance between any point in Q_k and any point of W_k is in I_{i-1}. Next, we replace each complete bipartite graph $Q_k \times W_k$ by a set E_k of expander graphs whose total number of edges is $O(|Q_k| + |W_k|)$. Then the total number of edges of all sets of expander graphs $\{E_k\}_k$ is $\sum_k O(|Q_k| + |W_k|) = O(n_i^{4/3} \log n_i)$. Each edge of an expander graph is associated with a distance of two points corresponding to the two nodes of the graph it connects. Let L_i denote the set of distances of all edges in all expander graphs of $\{E_k\}_k$; the size of L_i is $O(n_i^{4/3} \log n_i)$. Let $\mathcal{L}$ denote the union of all such L_i's. Then, $|\mathcal{L}| = \sum_{i=1}^{m} n_i^{4/3} \log n_i$, which is bounded by $O(n^{5/4} \log^{3/4} n)$ as discussed above. By doing binary search with the decision algorithm on $\mathcal{L}$, we can compute the smallest feasible value b_j and the largest infeasible value a_j of $\mathcal{L}$. Hence, $(a_j, b_j]$ contains r^* and (a_j, b_j) does not contain any value of $\mathcal{L}$. Note that when doing binary search on $\mathcal{L}$, we do not need to sort it first; instead we use the linear time selection algorithm [60]. As such, finding a_j and b_j can be done in $O(n^{5/4} \log^{3/4} n)$ time, which is also the total time of this round. Let $I_j = (a_j, b_j]$. The analysis of [30] shows that the total number of values of $\mathcal{D}$ in I_j is a constant fraction of the total number of values of $\mathcal{D}$ in I_{j-1}.

As the algorithm has $O(\log n)$ rounds and each round runs in $O(n^{5/4} \log^{3/4} n)$ time, the overall time of the algorithm is $O(n^{5/4} \log^{7/4} n)$. □

With the interval $(r'_1, r'_2]$ computed by the above lemma, we update $r_1 = \max\{r_1, r'_1\}$ and $r_2 = \min\{r_2, r'_2\}$. By definition, $r^* \in (r_1, r_2] \subseteq (r'_1, r'_2]$. Hence, the interval $(r_1, r_2]$ also has the same property as $(r'_1, r'_2]$ in Lemma 3.6.

Next, we run the BFS algorithm as in Section 3.3.2. To solve each instance of Subproblem 3.1, if one of the two involved cells is a large cell (we refer to this case as the *large-cell instance*), then we use the same algorithm as before, i.e., parametric search; otherwise (i.e., both involved cells are small cells; we refer to this case as *small-cell instance*), due to the preprocessing of Lemma 3.6, we can solve the subproblem directly using the original CS algorithm by picking an arbitrary value $r \in (r_1, r_2)$. In this way, the time for solving all small-cell instances in the entire BFS algorithm is $O(n)$. For each large-cell instance, it can be solved in $O(n \log n)$ time as discussed in Section 3.3.2. As the number of large cells

of $\mathcal{C}$ is at most $n^{1/4}\log^{3/4} n$ and $|N(C)| = O(1)$ for each cell $C \in \mathcal{C}$, the total number of large-cell instances of Subproblem 3.1 is at most $O(n^{1/4}\log^{3/4} n)$. Hence, the total time for solving the large-cell instances in the entire BFS algorithm is $O(n^{5/4}\log^{7/4} n)$. The proof of the following lemma presents the details of the new BFS algorithm sketched above.

Lemma 3.7. *The BFS algorithm, which computes r^*, can be implemented in $O(n^{5/4}\log^{7/4} n)$ time.*

Proof. We define S_i and $S_i(r)$ in the same way as in Section 3.3.2. Initially, we set $S_0 = \{s\}$. Before the i-step starts, we have an interval $(r_1, r_2]$. Again, the algorithm maintains an invariant that the i-th step shrinks $(r_1, r_2]$ so that it contains r^* and if $r^* \neq r_2$, then $S_i = S_i(r^*) = S_i(r)$ for any $r \in (r_1, r_2)$. Initially, the invariant trivially holds for S_0.

Consider the i-th step. Assume that the invariant holds for S_{i-1}, i.e., we have an interval $(r_1, r_2]$ containing r^* such that if $r^* \neq r_2$, then $S_{i-1} = S_{i-1}(r) = S_{i-1}(r^*)$ for any $r \in (r_1, r_2)$, and S_{i-1} is available to us. Using the grid information of $\Psi(P)$, we obtain the grid cells containing the points of S_{i-1}. For each such cell C, as before in Section 3.3.2, we add to S_i the points of $P \cap C$ that have not been discovered yet. Then, for each neighbor C' of C, we need to solve Subproblem 3.1; we use $\mathcal{I}$ to denote the set of instances of this subproblem in this step.

Consider two cells C and C' involved in an instance of $\mathcal{I}$. If one of them is a large cell, then we run the same parametric search algorithm as in Section 3.3.2, i.e., the three subroutines. As before, the time of the algorithm is bounded by $O(n \log n)$ and the algorithm shrinks the interval $(r_1, r_2]$ so that the algorithm invariant is maintained. Recall that in Section 3.3.2 we solve all problem instances in each step of the BFS algorithm all together. Here instead it suffices to solve each problem instance individually. As the number of large cells is at most $O(n^{1/4}\log^{3/4} n)$, the total number of large-cell instances in the entire BFS algorithm is $O(n^{1/4}\log^{3/4} n)$. Hence, the total time for solving the large-cell instances of Subproblem 3.1 in the entire BFS is $O(n^{5/4}\log^{7/4} n)$.

We now consider the small-cell instance where both C and C' are small cells. Note that in each instance of Subproblem 3.1, all red points are in one cell, say, C, and all blue

points are in the other cell C'. Let P_R be the set of red points in C and P_B be the set of blue points in C'. According to Lemma 3.6, if $r^* \neq r_2$ (and thus $r^* \in (r_1, r_2)$), then for any point $p \in P_R$ and any point $p' \in P_B$, either $\|p - p'\| < r$ holds for all $r \in (r_1, r_2)$ or $\|p - p'\| > r$ holds for all $r \in (r_1, r_2)$, implying that $\|p - p'\| > r^*$ if and only if $\|p - p'\| > r$ for any $r \in (r_1, r_2)$. Therefore, we can solve the subproblem in the following way. We first take any $r \in (r_1, r_2)$. Then we run the CS algorithm to solve the subproblem with r as the radius, which takes $O(n_r + n_b)$ time. Note that the interval $(r_1, r_2]$ will not be changed in this case. Due to the preprocessing in Lemma 3.6, the algorithm invariant still holds (i.e., $(r_1, r_2]$ contains r^* and if $r^* \neq r_2$, then $S_i = S_i(r^*) = S_i(r)$ for any $r \in (r_1, r_2)$). The total time for solving the small-cell instances in the entire BFS is $O(n)$ because as in the CS algorithm each cell will be involved in at most $O(1)$ instances of the subproblem in the entire BFS algorithm.

After the i-th step, as before, we obtain the set S_i and an interval $(r_1, r_2]$ containing r^* such that if $r^* \neq r_2$, then $S_i = S_i(r^*) = S_i(r)$ for any $r \in (r_1, r_2)$. If $t \in S_i$ and $i \leq \lambda$, then we can stop the algorithm; by Lemma 3.4, we have $r^* = r_2$. If $t \notin S_i$ and $i = \lambda$, we also stop the algorithm; by Lemma 3.5, we have $r^* = r_2$.

In summary, the overall time of the BFS algorithm is $O(n^{5/4} \log^{7/4} n)$. □

Combining with the algorithm of Lemma 3.6, the overall time of the algorithm for computing r^* is $O(n^{5/4} \log^{7/4} n)$. We thus obtain the following theorem.

Theorem 3.2. *The reverse shortest path problem for L_2 unweighted unit-disk graphs can be solved in $O(n^{5/4} \log^{7/4} n)$ time.*

3.5 The weighted case

We follow the notation introduced in Section 3.1 and Section 3.2, e.g., P, $G_r(P)$, $d_r(s, t)$, and r^*, but now defined for weighted unit-disk graphs. Our goal is to compute r^*. As discussed in Section 3.1.2, our algorithm utilizes parametric search by parameterizing the WX algorithm [1]. We begin with a review of the WX algorithm.

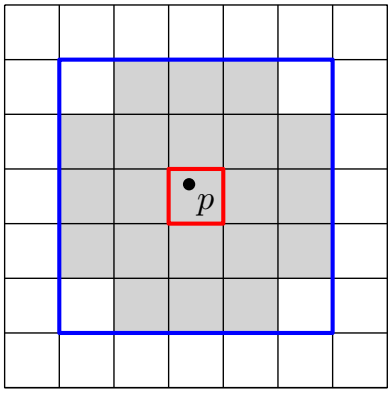


Fig. 3.8: The red cell that contains the point p is $\square_p$ and the square area bounded by blue segments is the patch $\boxplus_p$. All adjacent vertices of p in $G_r(P)$ must lie in the grey region.

3.5.1 A review of the WX algorithm

Given P, r, and a source point $s \in P$, the WX algorithm can compute shortest paths from s to all points of P in the weighted unit-disk graph $G_r(P)$, and the algorithm runs in $O(n \log^2 n)$ time.

For any point p in the plane, let $\odot_p$ denote the disk centered at p with radius r.

The first step is to implicitly build a grid $\Psi_r(P)$ of square cells whose side lengths are $r/\sqrt{2}$. For simplicity of discussion, we assume that every point of P lies in the interior of a cell of $\Psi_r(P)$. A *patch* of $\Psi_r(P)$ refers to a square area consisting of 5×5 cells. For a point $p \in P$, we use $\square_p$ to denote the cell of $\Psi_r(P)$ containing p and use $\boxplus_p$ to denote the patch whose central cell is $\square_p$ (e.g., see Fig. 3.8). We refer to cells of $\boxplus_p \setminus \square_p$ as the *neighboring cells* of $\square_p$. As the side length of each cell of $\Psi_r(P)$ is $r/\sqrt{2}$, any two points of P in a single cell of $\Psi_r(P)$ must be connected by an edge in $G_r(P)$. Moreover, if an edge connects two points p and q in $G_r(P)$, then q must lie in $\boxplus_p$ and vice versa. For any subset $Q \subseteq P$ and a cell $\square$ (resp.,a patch $\boxplus$) of $\Psi_r(P)$, define $Q_\square = Q \cap \square$ (resp., $Q_\boxplus = Q \cap \boxplus$). The step of implicitly building the grid actually computes the subset $P_\square$ for each cell $\square$ of $\Psi_r(P)$ that contains at least one point of P as well as associate pointers to each point $p \in P$ so that given any $p \in P$, the list of points of $P_{\square_p}$ (resp., $P_{\boxplus_p}$) can be accessed immediately. Building $\Psi_r(P)$ implicitly as above can be done in $O(n \log n)$ time, e.g., by the algorithm of Lemma 3.1.

The WX algorithm follows the basic idea of Dijkstra's algorithm and computes an array $dist[\cdot]$ for each point $p \in P$, where $dist[p]$ will be equal to $d_r(s, p)$ when the algorithm terminates. Different from Dijkstra's shortest path algorithm, which picks a single vertex in each iteration to update the shortest path information of other adjacent vertices, the WX algorithm aims to update in each iteration the shortest path information for all points within one single cell of $\Psi_r(P)$ and pass on the shortest path information to vertices lying in the neighboring cells.

A key subroutine used in the WX algorithm is UPDATE(U, V), which updates the shortest path information for a subset $V \subseteq P$ of points by using the shortest path information of another subset $U \subseteq P$ of points. Specifically, the subroutine finds, for each $v \in V$, $q_v = \arg\min_{u \in U \cap \bigodot_v} \{dist[u] + \|u - v\|\}$ and update $dist[v] = \min\{dist[v], dist[q_v] + \|q_v - v\|\}$.

With the subroutine UPDATE(U, V) in hand, the WX algorithm works as follows (refer to Algorithm 3.1 for the pseudocode).

Algorithm 3.1: The WX Algorithm [1]

```
1  Function WX(P, s):
2      for each p ∈ P do
3          dist[p] = ∞
4      end
5      dist[s] = 0
6      Q = P
7      while Q ≠ ∅ do
8          z = arg min_{p∈Q} {dist[p]}
9          UPDATE(Q_{⊞_z}, Q_{□_z}) // first update
10         UPDATE(Q_{□_z}, Q_{⊞_z}) // second update
11         Q = Q \ Q_{□_z}
12     end
13     return dist[·]
14 end
```

Initially, we set $dist[s] = 0$, $dist[p] = \infty$ for all other points $p \in P \setminus \{s\}$, and $Q = P$. Then we enter the main (while) loop. In each iteration, we find a point z with minimum $dist$-value from Q, and then execute two update subroutines UPDATE($Q_{\boxplus_z}$, $Q_{\square_z}$) and UP-

DATE($Q_{\square_z}$, $Q_{\boxplus_z}$). Next, points of $Q_{\square_z}$ are removed from Q, because it can be shown that $dist[p]$ for all points $p \in Q_{\square_z}$ have been correctly computed [1]. The algorithm stops once Q becomes $\emptyset$.

The efficiency of the algorithm hinges on the implementation of the two update subroutines. We give some details below, which are needed in our RSP algorithm as well.

The first update

For the first update UPDATE($Q_{\boxplus_z}$, $Q_{\square_z}$), the crucial step is finding a point $q_v \in Q_{\boxplus_z} \cap \bigodot_v$ for each point $v \in Q_{\square_z}$ such that $dist[q_v] + \|q_v - v\|$ is minimized. If we assign $dist[q]$ as a weight to each point $q \in Q_{\boxplus_z}$, then the problem is equivalent to finding the additively-weighted nearest neighbor q_v from $Q_{\boxplus_z} \cap \bigodot_v$ for each $v \in Q_{\square_z}$. To this end, Wang and Xue [1] proved a *key observation* that any point $q \in Q_{\boxplus_z}$ that minimizes $dist[q] + \|q - v\|$ must lie in $\bigodot_v$. This implies that for each point $v \in Q_{\square_z}$, its additively-weighted nearest neighbor in $Q_{\boxplus_z}$ is also its additively-weighted nearest neighbor in $Q_{\boxplus_z} \cap \bigodot_v$. As such, q_v for all $v \in Q_{\square_z}$ can be found by first building an additively-weighted Voronoi Diagram on points of $Q_{\boxplus_z}$ [57] and then performing point locations for all $v \in Q_{\square_z}$ [44, 45, 61]. In this way, since $\sum_{z_i} |P_{\boxplus_{z_i}}| = O(n)$, where z_i refers to the point z in the i-th iteration of the main loop, the first updates for all iterations of the main loop can be done in $O(n \log n)$ time in total [1].

The second update

The second update UPDATE($Q_{\square_z}$, $Q_{\boxplus_z}$) is more challenging because the above key observation no longer holds. Since $Q_{\boxplus_z}$ has $O(1)$ cells of $\Psi_r(P)$, it suffices to perform UPDATE($Q_{\square_z}, Q_{\square}$) for all cells $\square \in \boxplus_z$.

If $\square$ is $\square_z$, then $Q_{\square_z} = Q_{\square}$. Since the distance between any two points in $\square_z$ is at most r, we can easily implement UPDATE($Q_{\square_z}, Q_{\square}$) in $O(|Q_{\square_z}| \log |Q_{\square_z}|)$ time, by first building a additively-weighted Voronoi diagram on points of $Q_{\square_z}$ (each point $q \in Q_{\square_z}$ is assigned a weight equal to $dist[q]$), and then using it to find the additively-weighted nearest neighbor q_v for each point $v \in Q_{\square_z}$.

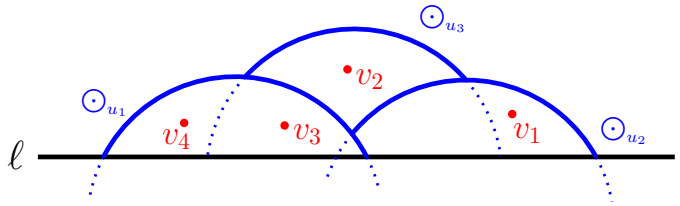


Fig. 3.9: Blue arcs are unit-disks centered at points $U = \{u_1, u_2, u_3\}$ which are sorted by their $dist[\cdot]$ values. We have $V_1 = \{v_3, v_4\}$, $V_2 = \{v_1\}$, and $V_3 = \{v_2\}$ in this example. Note that point v_3 is in unit-disk $\odot_{u_1}$ and $\odot_{u_3}$ at the same time, but v_3 is in subset $V_1 \subseteq V$ by the definition of V_i's, $1 \leq i \leq |U|$.

If $\square$ is not $\square_z$, a useful property is that $\square$ and $\square_z$ are separated by an axis-parallel line. The WX algorithm implements UPDATE$(Q_{\square_z}, Q_{\square})$ with the following three steps (see Fig. 3.9 for an example). Let $U = Q_{\square_z}$ and $V = Q_{\square}$.

1. Sort points of U as $\{u_1, u_2, ..., u_{|U|}\}$ such that $dist[u_1] \leq dist[u_2] \leq ... \leq dist[u_{|U|}]$.

2. Compute $|U|$ disjoint subsets $\{V_1, V_2, ..., V_{|U|}\}$ with $V_i = \{v \in V \mid v \in \odot_{u_i}$ and $v \notin \odot_{u_j}$ for all $1 \leq j < i\}$. Equivalently, for each point $v \in V$, v is in V_{i_v}, where i_v is the smallest index i (if exists) such that $\odot_{u_i}$ contains v.

3. Initialize $U' = \emptyset$. Proceed with $|U|$ iterations for $i = |U|, |U| - 1, ..., 1$ sequentially and do the following in each iteration for i: (1) Add u_i to U'; (2) for each point $v \in V_i$, compute $q_v = \arg\min_{u \in U'}\{dist[u] + \|u - v\|\}$; (3) update $dist[v] = \min\{dist[v], dist[q_v] + \|q_v - v\|\}$.

By the definition of V_i, $U \cap \odot_v \subseteq U' = \{u_{|U|}, u_{|U|-1}, ..., u_i\}$ for each $v \in V_i$ in the iteration for i of Step 3. Wang and Xue [1] proved that q_v found for each $v \in V_i$ in Step 3 must lie in $\odot_v$. They gave a method to implement Step 2 in $O(k \log k)$ time by making use of the property that U and V are separated by an axis-parallel line, where $k = |U| + |V|$. Step 3 can be considered as an offline insertion-only additively-weighted nearest neighbor searching problem and the WX algorithm solves the problem in $O(k \log^2 k)$ time using the standard logarithmic method [41], with $k = |U| + |V|$.

As such, the second updates for all iterations in the WX algorithm takes $O(n \log^2 n)$ time in total [1], which dominates the entire algorithm (other parts of the algorithm together takes $O(n \log n)$ time).

3.5.2 The RSP algorithm

We now tackle the RSP problem, i.e., given λ and $s, t \in P$, compute r^*. We will "parameterize" the WX algorithm reviewed above.

Recall that the decision problem is to decide whether $r^* \leq r$ for a given r. Notice that $r^* \leq r$ holds if and only if $d_r(s,t) \leq \lambda$. The decision problem can be solved in $O(n \log^2 n)$ time by running the WX algorithm on r. In the following, we refer to the WX algorithm as the *decision algorithm*. We say that r is a *feasible value* if $r^* \leq r$ and an *infeasible value* otherwise.

As discussed in Section 3.1.2, to find r^*, we run the decision algorithm with a parameter r in an interval $(r_1, r_2]$ by simulating the algorithm on the unknown r^*. The interval always contains r^* but will be shrunk during course of the algorithm (for simplicity, when we say $(r_1, r_2]$ is shrunk, this also include the case that $(r_1, r_2]$ does not change). Initially, we set $r_1 = 0$ and $r_2 = \infty$.

The first step is to build a grid for P. The goal is to shrink $(r_1, r_2]$ so that it contains r^* and if $r^* \neq r_2$ (and thus $r^* \in (r_1, r_2)$), for any $r \in (r_1, r_2)$, the grid $\Psi_r(P)$ has the same combinatorial structure as $\Psi_{r^*}(P)$ in the following sense: (1) Both grids have the same number of rows and columns; (2) for any point $p \in P$, p lies in the i-th row and j-th column of $\Psi_r(P)$ if and only if p lies in the i-th row and j-th column of $\Psi_{r^*}(P)$. This can be done by applying the algorithm in Lemma 3.2 but replacing the CS algorithm with the WX algorithm as the decision algorithm. The runtime becomes $O(n \log^3 n)$ because the WX algorithm runs in $O(n \log^2 n)$ time.

Let $(r_1, r_2]$ denote the interval after building the grid. We pick any $r \in (r_1, r_2)$ and compute the grid information of $\Psi_r(P)$, which has the same combinatorial structure as $\Psi_{r^*}(P)$ if $r^* \neq r_2$. Below, we will simply use $\Psi(P)$ to refer to the grid information computed above, meaning that it does not change with respect to $r \in (r_1, r_2)$.

We use $dist_r[\cdot]$, $Q(r)$, $z(r)$ respectively to refer to $dist[\cdot]$, Q, z in the WX algorithm running on a parameter r. We start with setting $dist_r[s] = 0$, $dist_r[p] = \infty$ for all $p \in P \backslash \{s\}$, and $Q(r) = P$.

Next we enter the main loop. As long as $Q(r) \neq \emptyset$, in each iteration, we will find a point $z(r)$ with the minimum $dist_r$-value from $Q(r)$ and update $dist_r$-values for points in $Q(r)_{\square_{z(r)}} \cup Q(r)_{\boxplus_{z(r)}}$. Points in $Q(r)_{\square_{z(r)}}$ are then removed from $Q(r)$. Each iteration will shrink $(r_1, r_2]$ such that the following algorithm invariant is maintained: $(r_1, r_2]$ contains r^* and if $r^* \neq r_2$, the following holds for all $r \in (r_1, r_2)$: $z(r) = z(r^*)$, $Q(r) = Q(r^*)$, and $dist_r[p] = dist_{r^*}[p]$ for all $p \in P$.

Consider an iteration of the main loop. We assume that the invariant holds before the iteration on the interval $(r_1, r_2]$, which is true before the first iteration. In the following, we describe our algorithm for the iteration and we will show that the invariant holds after the iteration. We assume that $r^* \neq r_2$. According to our invariant, for any $r \in (r_1, r_2)$, we have $z(r) = z(r^*)$, $Q(r) = Q(r^*)$, and $dist_r[p] = dist_{r^*}[p]$ for all $p \in P$.

We first find a point $z(r) \in Q(r)$ with the minimum $dist_r$-value. Since the invariant holds before the iteration, we have $z(r) = \arg\min_{p \in Q(r)} dist_r[p] = \arg\min_{p \in Q(r^*)} dist_{r^*}[p] = z(r^*)$.[2] Hence, no "parameterization" is needed in this step, i.e., all involved values in the computation of this step are independent of r.

Next, we perform the first update $\textsc{Update}(Q(r)_{\boxplus_{z(r)}}, Q(r)_{\square_{z(r)}})$. This step also does not need parameterization. Indeed, for each point $p \in Q(r)_{\boxplus_{z(r)}}$, we assign $dist_r[p]$ to p as a weight, and then construct the additively-weighted Voronoi diagram on $Q(r)_{\boxplus_{z(r)}}$. For each point $v \in Q(r)_{\square_{z(r)}}$, we use the diagram to find its additively-weighted nearest neighbor $q_v(r) \in Q(r)_{\boxplus_{z(r)}}$ and update $dist_r[v] = \min\{dist_r[v], dist_r[q_v(r)] + \|q_v(r) - v\|\}$. Since $z(r) = z(r^*)$, and $Q(r) = Q(r^*)$, we have $Q(r)_{\boxplus_{z(r)}} = Q(r^*)_{\boxplus_{z(r^*)}}$ and $Q(r)_{\square_{z(r)}} = Q(r^*)_{\square_{z(r^*)}}$. Further, since $dist_r[p] = dist_{r^*}[p]$ for all $p \in P$, for each point $v \in Q(r)_{\square_{z(r)}}$, $q_v(r) = q_v(r^*)$ and each updated $dist_r[v]$ in our algorithm is equal to the corresponding

[2] When picking $z(r)$, we break ties following the same way as the WX algorithm. This guarantees $z(r) = z(r^*)$ even if ties happen.

updated $dist_{r^*}[v]$ in the same iteration of the WX algorithm running on r^*. As such, the invariant still holds after the first update.

Implementing the second update $\textsc{Update}(Q(r)_{\square_{z(r)}}, Q(r)_{\boxplus_{z(r)}})$ is more challenging and parameterization is necessary. It suffices to implement $\textsc{Update}(Q(r)_{\square_{z(r)}}, Q(r)_{\square})$ for all cells $\square \in \boxplus_{z(r)}$.

If $\square$ is $\square_{z(r)}$, then $Q(r)_{\square_{z(r)}} = Q(r)_{\square}$. In this case, again no parameterization is needed. Since the distance between any two points in $\square_{z(r)}$ is at most r, we can easily implement $\textsc{Update}(Q(r)_{\square_{z(r)}}, Q(r)_{\square})$ in $O(|Q(r)_{\square_z(r)}| \log |Q(r)_{\square_z(r)}|)$ time, by first building a additively-weighted Voronoi diagram on points of $Q(r)_{\square_{z(r)}}$ (each point $p \in Q(r)_{\square_{z(r)}}$ is assigned a weight equal to $dist_r[p]$), and then using it to find the additively-weighted nearest neighbor $q_v(r)$ for each point $v \in Q(r)_{\square_z}$. By an analysis similar to the above first update, the invariant still holds.

We now consider the case where $\square$ is not $\square_{z(r)}$. In this case, $\square$ and $\square_{z(r)}$ are separated by an axis-parallel line ℓ. Without loss of generality, we assume that ℓ is horizontal and $\square_{z(r)}$ is below ℓ. Since $z(r) = z(r^*)$ and $Q(r) = Q(r^*)$ for all $r \in (r_1, r_2)$, we let $U = Q(r)_{\square_{z(r)}}$ and $V = Q(r)_{\square}$, meaning that both U and V are independent of $r \in (r_1, r_2)$. Recall that there are three steps in the second update of the decision algorithm. Our algorithm needs to simulate all three steps. As will be seen later, only the second step needs parameterization.

The first step is to sort points in U by their $dist_r$-values. Since $dist_r[p] = dist_{r^*}[p]$ for all $p \in P$, the sorted list $\{u_1, u_2, ..., u_{|U|}\}$ of U obtained in our algorithm is the same as the sorted list obtained in the decision algorithm running on r^*.

For any r, we use $\bigodot_p(r)$ to denote the disk centered at a point p with radius r.

The second step is to compute $|U|$ disjoint subsets $\{V_1(r), V_2(r), ..., V_{|U|}(r)\}$ of V such that $V_i(r) = \{v \mid i_v(r) = i, v \in V\}$, where $i_v(r)$ is the smallest index such that $\bigodot_{u_{i_v(r)}}(r)$ contains point v. This step needs parameterization. We will shrink the interval $(r_1, r_2]$ so that it still contains r^* and if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, $V_i(r) = V_i(r^*)$ holds for all $1 \leq i \leq |U|$ (it suffices to ensure $i_v(r) = i_v(r^*)$ for all $v \in V$). Our algorithm relies on the following observation, which is based on the definition of $i_v(r)$.

Observation 3.1. *For any point $v \in V$, if $\bigodot_{u_j}(r)$ contains v with $1 \leq j \leq |U|$, then $i_v(r) \leq j$.*

For a subset $P' \subseteq P$, let $\mathcal{F}_r(P')$ denote the union of the disks centered at points of P' with radius r. We first solve a subproblem in the following lemma.

Lemma 3.8. *Suppose $(r_1, r_2]$ contains r^* such that if $r^* \neq r_2$, then for all $r \in (r_1, r_2)$, $dist_r[p] = dist_{r^*}[p]$ for all points $p \in P$. For a subset $U' \subseteq U$ and a subset $V' \subseteq V$, in $O(n \log^2 n \cdot \log(|U'| + |V'|))$ time we can shrink $(r_1, r_2]$ so that it still contains r^* and if $r^* \neq r_2$, then for all $r \in (r_1, r_2)$, for any $v \in V'$, v is contained in $\mathcal{F}_r(U')$ if and only if v is contained in $\mathcal{F}_{r^*}(U')$.*

Proof. Recall that all points of U are below ℓ and all points of V are above ℓ. For any r, the problem to determine whether v is contained in $\mathcal{F}_r(U')$ for each $v \in V'$ is an instance of Subproblem 3.1 (i.e., consider the points of U' as red points and the points of V' as blue points). Recall that solving Subproblem 3.1 for a fixed r involves three subroutines and we also give a parameterized algorithm for solving it on the unknown r^* in Section 3.3.2 for the unweighted case. Here, to achieve the lemma, we can essentially apply the same algorithm as in Section 3.3.2 but instead use the WX algorithm as the decision algorithm. We sketch it below.

Let $\mathcal{U}_r(U')$ denote the upper envelope of the portions of the disks $\bigodot_u(r)$ above ℓ for all $u \in U'$. A point $v \in V'$ is in $\mathcal{F}_r(U')$ if and only if v is below $\mathcal{U}_r(U')$. The algorithm has three subroutines. The first subroutine is to shrink $(r_1, r_2]$ so that it still contains r^* and if $r^* \neq r_2$, then for all $r \in (r_1, r_2)$, $\mathcal{U}_r(U')$ has the same combinatorial structure as $\mathcal{U}_{r^*}(U')$. This can be done by applying the algorithm of Section 3.3.2 but using the WX algorithm as the decision algorithm. The second subroutine is to shrink $(r_1, r_2]$ such that it still contains r^* and if $r^* \neq r_2$, then for all $r \in (r_1, r_2)$, the sorted list of the vertices of $\mathcal{U}_r(U')$ and all points of V' is the same as the sorted list of the vertices of $\mathcal{U}_{r^*}(U')$ and all points of V'. This can be done by applying the algorithm of Section 3.3.2 but using the WX algorithm as the decision algorithm. The third subroutine is to shrink $(r_1, r_2]$ so that $(r_1, r_2]$ contains r^* and if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, for any $v \in V'$, v is below

the arc spanning it in $\mathcal{U}_r(U')$ if and only if v is below the arc spanning it in $\mathcal{U}_{r^*}(U')$. This can be done by applying the algorithm of Section 3.3.2 but using the WX algorithm as the decision algorithm. Following the analysis of Sections 3.3.2, 3.3.2, and 3.3.2, the total time of the algorithm is bounded by $O(n \log^2 n \cdot \log(|U'| + |V'|))$ because the decision algorithm runs in $O(n \log^2 n)$ time (and both $|U'|$ and $|V'|$ are no more than n). □

Recall that we have an interval $(r_1, r_2]$. Our goal is to shrink it so that it still contains r^* and if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, $V_i(r) = V_i(r^*)$ holds for all $1 \leq i \leq |U|$. Based on Observation 3.1 and using Lemma 3.8, we have the following lemma.

Lemma 3.9. *We can shrink the interval $(r_1, r_2]$ in $O(n \log^4 n)$ time so that it still contains r^* and if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, $V_i(r) = V_i(r^*)$ holds for all $1 \leq i \leq |U|$.*

Proof. To have $V_i(r) = V_i(r^*)$ for all $1 \leq i \leq |U|$, it suffices to ensure $i_v(r) = i_v(r^*)$ for all points $v \in V$. Let $M = |U|$ and $N = |V|$. Note that $M \leq n$ and $N \leq n$.

As defined in the proof of Lemma 3.8, for any subset $U' \subseteq U$ and any r, we use $\mathcal{U}_r(U')$ to denote the upper envelope of the portions of $\bigodot_u(r)$ above ℓ for all $u \in U'$.

In light of Observation 3.1, we use the divide and conquer approach. Recall that $U = \{u_1, u_2, \ldots, u_M\}$. Consider the following subproblem on (U, V): shrink $(r_1, r_2]$ so that it still contains r^* and if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, the following holds, for any $v \in V$, v is below $\mathcal{U}_r(U_1)$ if and only if v is below $\mathcal{U}_{r^*}(U_1)$, where U_1 is the first half of U, i.e., $U_1 = \{u_1, u_2, ..., u_{\lfloor \frac{M}{2} \rfloor}\}$. The subproblem can be solved in $O(n \log^3 n)$ time by applying Lemma 3.8. Next, we pick any $r \in (r_1, r_2)$ and compute $\mathcal{U}_r(U_1)$ and find the subset V_1 of the points of V that are below $\mathcal{U}_r(U_1)$ (e.g., see Fig. 3.10). By Observation 3.1, for each point $v \in V$, $i_v(r) \leq \lfloor \frac{M}{2} \rfloor$ if $v \in V_1$ and $i_v(r) > \lfloor \frac{M}{2} \rfloor$ otherwise. By the above property of $(r_1, r_2]$, for each point $v \in V$, we also have $i_v(r^*) \leq \lfloor \frac{M}{2} \rfloor$ if $v \in V_1$ and $i_v(r^*) > \lfloor \frac{M}{2} \rfloor$ otherwise.

We have determined whether $i_v(r^*) \leq \lfloor \frac{M}{2} \rfloor$ for each point $v \in V$ after the first call of Lemma 3.8 as discussed above. To shrink the range of $i_v(r^*)$ for each $v \in V$ further, we construct two subproblems for sets V_1 and $V \setminus V_1$ with their corresponding subsets of U. More specifically, we solve two subproblems recursively: one on (U_1, V_1) and the other on

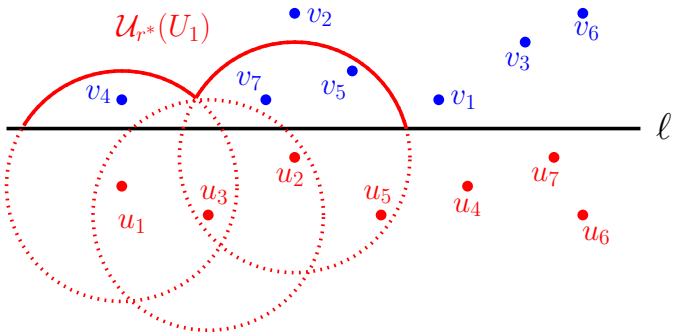


Fig. 3.10: Illustrating U_1 and V_1, where $U_1 = \{u_1, u_2, u_3\}$ and $V_1 = \{v_4, v_5, v_7\}$. The solid arcs are on $\mathcal{U}_{r^*}(U_1)$.

$(U \setminus U_1, V \setminus V_1)$. Both subproblems use $(r_1, r_2]$ as their "input intervals" and solving each subproblem will produce a new shrunk "output interval" $(r_1, r_2]$. Consider a subproblem on (U', V') with $U' \subseteq U$ and $V' \subseteq V$. If $|U'| = 1$, then we solve this problem "directly" (i.e., this is the base case) as follows. Assume that $r^* \neq r_2$ and let r be any value in (r_1, r_2). Let u_j be the only point of U'. There are two cases depending on the index j of point $u_j \in U'$. If $j < M = |U|$ (i.e., u_j is not the last point of the sorted list of points in set U), according to our algorithm and based on Observation 3.1, $i_v(r) = i_v(r^*) = j$ holds for all points $v \in V'$. If $j = M$, however, for each point $v \in V'$, it is possible that v is not contained in $\bigodot_u(r^*)$ for any point $u \in U$, in which case v is not below $\mathcal{U}_{r^*}(U)$ and thus is not below $\mathcal{U}_{r^*}(U')$. On the other hand, if v is below $\mathcal{U}_{r^*}(U')$, then $i_v(r^*) = M$. To solve the case of $j = M$, we can simply apply Lemma 3.8 on U' and V', after which we obtain an interval $(r_1, r_2]$. Then, we pick any $r \in (r_1, r_2)$ and for any $v \in V'$ with v contained in $\bigodot_{u_M}(r)$, $i_v(r) = i_v(r^*) = M$ holds if $r^* \neq r_2$.

The above divide-and-conquer algorithm can be viewed as a binary tree structure T in which each node represents a subproblem. The input of the subproblem for each node is derived from the result of solving the subproblem represented by its parent node. We shrink each $i_v(r^*)$ for $v \in V$ to a specific value in the end (i.e., subproblems corresponding to leaves of this binary tree T). Clearly, the height of T is $O(\log M)$ and T has $\Theta(M)$ nodes. If we solve each subproblem individually by Lemma 3.8 as described above, then the algorithm would take $\Omega(Mn)$ time because there are $\Omega(M)$ subproblems and solving each

subproblem by Lemma 3.8 takes $\Omega(n)$ time, which would result in an $\Omega(n^2)$ time algorithm in the worst case. To reduce the runtime, instead, we solve subproblems at the same level of T simultaneously (or "in parallel") by applying the algorithm of Lemma 3.8, as follows.

Consider all subproblems in the same level of T; let S denote the set of all these subproblems. There is an input interval $(r_1, r_2]$ for all subproblems of S, which is true initially at the root for (U, V). After solving all subproblems in this level, our algorithm will produce a single shrunk interval $(r_1, r_2]$, which will be used as the input interval for all subproblems in the next level of T.

Recall that the algorithm of Lemma 3.8 has three subroutines (which follow the algorithm in Section 3.3.2), each of which involves computing a set of critical values and then performing binary search on them using the decision algorithm to shrink the interval $(r_1, r_2]$. To solve all subproblems of S simultaneously using the algorithm of Lemma 3.8, our idea is that in each of the three subroutines, we perform binary search on the critical values of all subproblems of S (this again follows the same way as in Section 3.3.2, where critical values of all instances of $\mathcal{I}$ are considered all together), i.e., we solve all these subproblems "in parallel". In this way, solving all subproblems of S together only needs to call the decision algorithm $O(\log n)$ times. The details are given below.

For the first subroutine, the goal is to determine the combinatorial structure of the upper envelope. The critical values in all three subroutines are defined as in Section 3.3.2. For each subproblem on (U', V'), we compute the Voronoi diagram for U' and then find the critical values. Notice that the subsets U' (resp., V') for all subproblems of S form a partition of U (resp., V), and thus the total time for building the diagram and computing the critical values for all subproblems of S takes $O((M + N) \log(M + N))$ time in total. Also, the total number of critical values is $O(N)$. Performing the binary search on these critical values as before can be done in $O(n \log^2 n \cdot \log N)$ time, after which we obtain a shrunk interval $(r_1, r_2]$. This finishes the first subroutine for all subproblems of S, which takes $O(n \log^3 n)$ time (since $M \leq n$ and $N \leq n$).

The second subroutine is to sort all points of V' in each subproblem on (U', V') along

with the vertices of the upper envelope $\mathcal{U}_{r^*}(U')$. We now put all involved points of all subproblems of S in one coordinate system and sort them all together (in the same way as in Section 3.3.2). Since the subsets V' (resp., U') of all subproblems of S form a partition of V (resp., U), the total number of points in the subsets V' in all subproblems of S is N. Also, the number of vertices of $\mathcal{U}_{r^*}(U')$ is proportional to $|U'|$. Hence, the total number of vertices of the upper envelopes $\mathcal{U}_{r^*}(U')$ in all subproblems of S is $O(M)$. As such, the total number of points we need to sort is $O(M+N)$. We apply the same algorithm as before to sort them, i.e., Cole's parametric search [51] with AKS sorting network [59] and our decision algorithm. Sorting all involved points can be done in $O(n\log^2 n\cdot\log(M+N))$ time, after which a shrunk interval $(r_1,r_2]$ is obtained. This finishes the second subroutine for all subproblems of S, which takes $O(n\log^3 n)$ time.

For the third subroutine, we collect the critical values in each subproblem of S in the same way as before. The total number of critical values for all subproblems is N. We perform binary search on these critical values in the same way as before, after which a shrunk interval $(r_1,r_2]$ is obtained. The total time is $O(n\log^2 n\cdot\log N)$. This finishes the third subroutine for all subproblems, which takes $O(n\log^3 n)$ time. The final interval $(r_1,r_2]$ will be used as the input interval for all subproblems in the next level of T.

In summary, solving all subproblems in the same level of T can be done in $O(n\log^3 n)$ time. As T has $O(\log M)$ levels, the total time of the overall algorithm is $O(n\log^4 n)$. □

With Lemma 3.9, we obtain subsets $\{V_1(r),V_2(r),...,V_{|U|}(r)\}$ and an interval $(r_1,r_2]$ containing r^* such that if $r^*\neq r_2$, for any $r\in(r_1,r_2)$, $V_i(r)=V_i(r^*)$ holds for all $1\leq i\leq|U|$. Note that neither the array $dist_r[\cdot]$ nor $Q(r)$ is modified during the algorithm of Lemma 3.9. Hence, if $r^*\neq r_2$, for all $r\in(r_1,r_2]$, we still have $Q(r)=Q(r^*)$ and $dist_r[p]=dist_{r^*}[p]$ for all points $p\in P$. Thus, our algorithm invariant still holds. This finishes the second step of the second update.

The third step of the second update is to solve the offline insertion-only additively-weighted nearest neighbor searching problem. This step does not need parameterization. Similar to the first update, we pick any $r\in(r_1,r_2)$ and apply the WX algorithm directly.

Indeed, the algorithm on r^* only relies on the following information: U and its sorted list by $dist_{r^*}[\cdot]$ values and the subsets $V_1(r^*), \ldots, V_{|U|}(r^*)$. Recall that if $r^* \neq r_2$, then for all $r \in (r_1, r_2)$, $dist_r[p] = dist_{r^*}[p]$ for all $p \in P$, and $V_i(r) = V_i(r^*)$ for all $1 \leq i \leq |U|$. As such, if we pick any $r \in (r_1, r_2)$ and apply the WX algorithm directly, $dist_r[v] = dist_{r^*}[v]$ holds for all points $v \in V$ after this step. Therefore, as in the WX algorithm, this step can be done in $O(k \log^2 k)$ time, where $k = |U| + |V|$.

This finishes the second update of the algorithm. As discussed above, the algorithm invariant holds for the interval $(r_1, r_2]$.

The final step of the iteration is to remove points in $Q(r)_{\square_{z(r)}}$ from $Q(r)$. Since if $r^* \neq r_2$, for all $r \in (r_1, r_2)$, $Q(r) = Q(r^*)$, $z(r) = z(r^*)$, and $Q(r)_{\square_{z(r)}} = Q(r^*)_{\square_{z(r^*)}}$, $Q(r) = Q(r^*)$ still holds after this point removal operation. Therefore, our algorithm invariant holds after the iteration.

In summary, each iteration of our algorithm takes $O(n \log^4 n)$ time. If the point t is contained in $\square_{z(r)}$ (i.e., t is reached) in the current iteration, then we terminate the algorithm. The following lemma shows that we can simply return r_2 as r^*.

Lemma 3.10. *Suppose that t is contained in $\square_{z(r)}$ in an iteration of our algorithm and $(r_1, r_2]$ is the interval after the iteration. Then $r^* = r_2$.*

Proof. Assume to the contrary that $r^* \neq r_2$. Then we have $r^* \in (r_1, r_2)$ since $r^* \in (r_1, r_2]$. Let $r' = (r_1 + r^*)/2$, and thus $r' \in (r_1, r_2)$ and $r' < r^*$. By our algorithm invariant and the correctness of the WX algorithm ($dist_r[p] = d_r(s, p)$ for all points $p \in P_{\square_{z(r)}}$ after the iteration), we have $d_{r'}(s, t) = dist_{r'}[t] = dist_{r^*}[t] = d_{r^*}(s, t)$. By the definition of r^*, $d_{r^*}(s, t) \leq \lambda$. Therefore, $d_{r'}(s, t) \leq \lambda$. But this contradicts with the definition of r^* since $r^* = \arg\min_r \{d_r(s, t) \leq \lambda\}$. The lemma thus holds. □

The algorithm may take $\Omega(n^2)$ time because t may be reached in $\Omega(n)$ iterations. A further improvement is discussed in the next subsection.

3.5.3 A further improvement

To further reduce the runtime of the algorithm, we borrow a technique from Section 3.4 to partition the cells of the grid into large and small cells.

As before, we first compute the grid information $\Psi(P)$ and obtain an interval $(r_1, r_2]$. Let $\mathcal{C}$ denote the set of all non-empty cells of $\Psi(P)$ (i.e., cells that contain at least one point of P). For each cell $C \in \mathcal{C}$, let $N(C)$ denote the set of non-empty neighboring cells of C in $\mathcal{C}$ and $P(C)$ the set of points of P contained in cell C. We have $|N(C)| = O(1)$ and $|\mathcal{C}| = O(n)$. A cell C of $\mathcal{C}$ is a *large cell* if it contains at least $n^{3/4}\log^{3/2} n$ points of P, i.e., $|P(C)| \geq n^{3/4}\log^{3/2} n$, and a *small cell* otherwise. Clearly, $\mathcal{C}$ has at most $n^{1/4}/\log^{3/2} n$ large cells. For all pairs of non-empty neighboring cells (C, C'), with $C \in \mathcal{C}$ and $C' \in N(C)$, (C, C') is a *small-cell pair* if both C and C' are small cells, and a *large-cell pair* otherwise, i.e., at least one cell is a large cell. Since $N(C) = O(1)$ for each cell $C \in \mathcal{C}$, there are $O(n^{1/4}/\log^{3/2} n)$ large-cell pairs.

We follow the algorithmic framework in Section 3.4. Notice that in each iteration of the main loop in our previous algorithm, only the second step of the second update parameterizes the WX algorithm (i.e., the decision algorithm is called on certain critical values); in that step, we need to process $O(1)$ pairs of cells (C, C') with $C \in \mathcal{C}$ and $C' \in N(C)$. No matter how many points of P are contained in the two cells, we need $O(n\log^4 n)$ time to perform the parametric search due to Lemma 3.9. To reduce the time, we preprocess all small-cell pairs so that the algorithm only needs to perform the parametric search for large-cell pairs. Since there are only $O(n^{1/4}/\log^{3/2} n)$ large-cell pairs, the total time we spend on parametric search can be reduced to $O(n^{5/4}\log^{5/2} n)$. For those small-cell pairs, the preprocessing provides sufficient information to allow us to simply run the original WX algorithm without parametric search. Specifically, before we enter the main loop of the algorithm (and after the grid information $\Psi(P)$ is computed, along with an interval $(r_1, r_2]$), we preprocess all small-cell pairs using the following lemma.

Lemma 3.11. *In $O(n^{5/4}\log^{5/2} n)$ time we can shrink the interval $(r_1, r_2]$ so that it still contains r^* and if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, for any small-cell pair (C, C') with*

$C \in \mathcal{C}$ and $C' \in N(C)$, an edge connects a point $p \in P(C)$ and a point $p' \in P(C')$ in $G_r(P)$ if and only if an edge connects p and p' in $G_{r^*}(P)$.

Proof. Lemma 3.6 essentially solves the same problem for the unweighted case. Here we follow the same algorithm as in Lemma 3.6 but replace their decision algorithm by our decision algorithm for the weighted case. The algorithm has $O(\log n)$ iterations, and following the same analysis as in Lemma 3.6 and using the new threshold $n^{3/4}\log^{3/2} n$ for defining large cells, one can show that each iteration takes $O(n^{5/4}\log^{3/2} n)$ time. More specifically, if we use the same notation as in the proof of Lemma 3.6, then we have $n_i \leq 2 \cdot n^{3/4}\log^{3/2} n$, and thus $|\mathcal{L}| = \sum_{i=1}^{m} n_i^{4/3} \log n_i$ is bounded by $O(n^{5/4}\log^{3/2} n)$. Therefore, the total running time of the algorithm is $O(n^{5/4}\log^{5/2} n)$. □

Let $(r_1, r_2]$ denote the interval obtained after the preprocessing for all small-cell pairs in Lemma 3.11. Lemma 3.11 essentially guarantees that if $r^* \neq r_2$, then for any $r \in (r_1, r_2)$, the adjacency relation of points in any small-cell pair in $G_r(P)$ is the same as that in $G_{r^*}(P)$. Note that if $(r_1, r_2]$ is shrunk so that it still contains r^*, then the above property still holds for the shrunk interval. Based on this property, combining with our previous algorithm, we have the following theorem.

Theorem 3.3. *The reverse shortest path problem for L_2 weighted unit-disk graphs can be solved in $O(n^{5/4}\log^{5/2} n)$ time.*

Proof. The goal is to compute r^*. We first build a grid $\Psi(P)$ along with an interval $(r_1, r_2]$ in $O(n\log^3 n)$ time. Then we classify all non-empty cells in $\Psi(P)$ to large cells and small cells. Next, we use Lemma 3.11 to shrink the interval $(r_1, r_2]$ in $O(n^{5/4}\log^{5/2} n)$ time.

We proceed to the main loop of the algorithm. In each iteration, we proceed in the same way as before except that the second step of the second update $\textsc{Update}(Q(r)_{\square_{z(r)}}, Q(r)_{\boxplus_{z(r)}})$ is now executed as follows. Recall that it suffices to perform $\textsc{Update}(Q(r)_C, Q(r)_{C'})$ with $C = \square_{z(r)}$ and $C' \in N(C)$. If (C, C') is a large-cell pair, then we apply our parametric search procedure in the same way as before. Since the number of large-cell pairs is $O(n^{1/4}/\log^{3/2} n)$ and implementing the second step of $\textsc{Update}(Q(r)_C, Q(r)_{C'})$ with the

parametric search takes $O(n \log^4 n)$ time by Lemma 3.9. Thus the total time we spend on all large-cell pairs is $O(n^{5/4} \log^{5/2} n)$. If (C, C') is a small-cell pair, according to the property of $(r_1, r_2]$ in the statement of Lemma 3.11, we can simply pick any value $r \in (r_1, r_2)$ and then apply the WX algorithm directly. Following the time complexity of the WX algorithm, the second step of UPDATE$(Q(r)_C,\ Q(r)_{C'})$ of all small-cell pairs (C, C') together takes $O(n \log n)$ time. The remaining parts of our algorithm together take the same running time as the WX algorithm, which is $O(n \log^2 n)$.

We thus conclude that the total time of our algorithm is bounded by $O(n^{5/4} \log^{5/2} n)$.

□

3.6 Concluding remarks

In this chapter, we propose two algorithms for the RSP problem in unweighted unit-disk graphs with time complexities of $O(\lfloor \lambda \rfloor \cdot n \log n)$ and $O(n^{5/4} \log^{7/4} n)$, respectively. We also give an algorithm for the RSP problem in weighted unit-disk graphs with a time complexity of $O(n^{5/4} \log^{5/2} n)$. Interestingly, our second unweighted RSP algorithm and the weighted RSP algorithm break the $O(n^{4/3})$ time barrier for certain geometric problems [62, 63].

Our RSP problem is defined with respect to a pair of points (s, t). Our techniques can be extended to solve a more general "single-source" version of the problem: Given a source point $s \in P$ and a value λ, compute the smallest value r^* such that the lengths of shortest paths from s to all vertices of $G_r(P)$ are at most λ, i.e., $\max_{t \in P} d_{r^*}(s, t) \leq \lambda$. The decision problem (i.e., deciding whether $r \geq r^*$ for any r) now becomes deciding whether $\max_{t \in P} d_r(s, t) \leq \lambda$. The algorithm of Chan and Skrepetos [13], the algorithm of Wang and Xue [1], and the algorithm of Wang and Zhao [14] are actually for finding shortest paths from s to all vertices of $G_r(P)$. Thus we can solve the decision problem by using the algorithm of Chan and Skrepetos [13] for the unweighted case, and the algorithm of Wang and Xue [1] for the weighted case. As such, to compute r^*, we can follow the same algorithm scheme as before but instead use the above new decision algorithm. In addition, for the unweighted case, we make the following changes to the first algorithm (the second algorithm is changed accordingly). After the i-th step of the BFS, which computes a set S_i

along with an interval $(r_1, r_2]$. If all points of P have been discovered after this step and $i \leq \lfloor \lambda \rfloor$, then we have $r^* = r_2$ and stop the algorithm; the proof is similar to Lemma 3.4. We also stop the algorithm with $r^* = r_2$ if $i = \lfloor \lambda \rfloor$ and not all points of P have been discovered; the proof is similar to Lemma 3.5. As before, the algorithm will stop in at most $\lfloor \lambda \rfloor$ steps. In this way, the first algorithm can compute r^* in $O(\lfloor \lambda \rfloor \cdot n \log n)$ time. Analogously, the second algorithm can compute r^* in $O(n^{5/4} \log^{7/4} n)$ time. For the weighted case, our original algorithm terminates once t is reached but now we instead halt the algorithm once all points of P are reached, which does not affect the running time asymptotically. As such, the "single-source" version of the weighted RSP problem can be solved in $O(n^{5/4} \log^{5/2} n)$ time.

CHAPTER 4

COMPUTING THE MINIMUM BOTTLENECK MOVING SPANNING TREE

4.1 Introduction

We consider the computation of the Euclidean minimum bottleneck moving spanning tree for a set of moving points in the plane. The results in this chapter have been published in a conference [21].

4.1.1 Problem definitions

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 [64] 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 chapter, motivated by visualizations of time-varying spatial data [20], 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. 4.1; 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

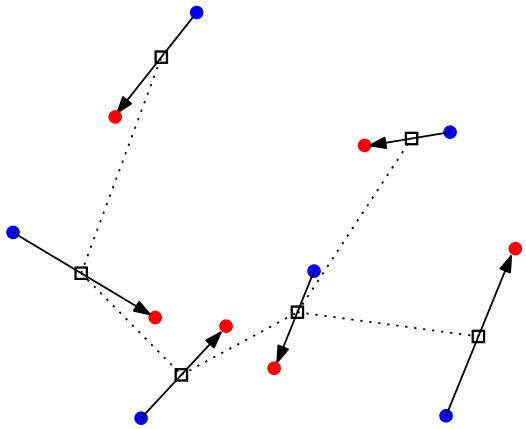


Fig. 4.1: Each pair of red and blue points connected by a black arrow represents a moving point. Blue points denote locations at $t = 0$ and red points are locations at $t = 1$. Black boxes are locations of these moving points at certain time and the dashed segments form a spanning tree.

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 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 chapter, 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 [20]. To solve the problem, the authors of [20] 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 [20]). Using the above property, the authors of [20] 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 of [20] 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 [65], the entire algorithm for computing T^* runs in $O(n^2)$ time in total [20].

4.1.2 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 [20], 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 $b(T^*) \leq \lambda$. 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 $|pq| \leq \lambda$ (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. 4.2). 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 [30] 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 $|pq| \leq \lambda$,

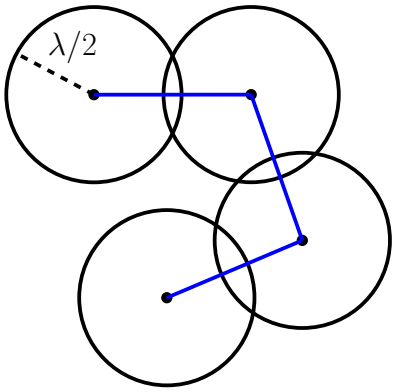


Fig. 4.2: Illustrating a unit-disk graph. Two points are connected (by a blue segment) if their distance is less than or equal to λ. In other words, two points are connected if congruent disks centered at them with radius $\lambda/2$ intersect.

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 [30], 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 [30], 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.

4.1.3 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 [20] 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 [20].

Geometric problems for moving objects have been studied extensively in the literature, e.g., [66,67]. In particular, kinetic data structures were proposed to maintain the minimum spanning tree for moving points in the plane [66,68]. 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 [67].

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 [9, 69–71]. Applying Chan's recent work [72] 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., [73–76]. In addition, Wang [33] 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 [77–79].

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 [1,6,7,13,14,17,18], clique [11], independent set [12], diameter [7,8,13], 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 [1,6]. 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 4.2. The algorithm uses our data structure for the deletion-only unit-disk range emptiness query problem, which is given in Section 4.3. Section 4.4 concludes.

4.2 Algorithm for moving-EMBST

We follow the notation in Section 4.1, 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 4.1.2, 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 4.2.1 and the algorithm for the optimization problem is described in Section 4.2.2.

4.2.1 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 [20].

Observation 4.1. ([20]) $\max_{t\in[0,1]} |p(t)q(t)| = \max\{|p(0)q(0)|, |p(1)q(1)|\}$ *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 $G_\lambda(1)$.

Lemma 4.1. *Given any $\lambda > 0$, $b(T^*) \leq \lambda$ if and only if $G_\lambda(0)$ and $G_\lambda(1)$ have a common spanning tree.*

Proof. Suppose $G_\lambda(0)$ and $G_\lambda(1)$ have a common spanning tree T in G_λ. Then for any edge of T connecting two points $p, q \in P$, since the edge appears in both $G_\lambda(0)$ and $G_\lambda(1)$, it holds that $|p(0)q(0)| \leq \lambda$ and $|p(1)q(1)| \leq \lambda$, and thus $\max\{|p(0)q(0)|, |p(1)q(1)|\} \leq \lambda$. By Observation 4.1, we have $b(T) = \max_{t\in[0,1]} b_T(t) \leq \lambda$. Since $b(T^*) \leq b(T)$ by the definition of T^*, we obtain $b(T^*) \leq \lambda$.

Now suppose $b(T^*) \leq \lambda$. We argue that T^* must be a common spanning tree of $G_\lambda(0)$ and $G_\lambda(1)$. Indeed, since $b(T^*) = \max_{t\in[0,1]} b_{T^*}(t) \leq \lambda$, for any edge of T^* connecting two points $p, q \in P$, $|p(t)q(t)| \leq \lambda$ for any $t \in [0, 1]$, and in particular, $|p(0)q(0)| \leq \lambda$ and $|p(1)q(1)| \leq \lambda$, implying that $G_\lambda(0)$ has an edge connecting p and q and so does $G_\lambda(1)$. As such, T^* must be a common spanning tree of $G_\lambda(0)$ and $G_\lambda(1)$. □

In light of Lemma 4.1, 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 4.3.

Theorem 4.1. *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 [30] 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 4.1 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 [30], we have the following lemma (which is essentially Theorem 3.3 in [30]) for computing a *compact representation* of all pairs (p, q) of points of P with $|p(0)q(0)| \leq \lambda$.

Lemma 4.2. (Theorem 3.3 [30]) *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 4.2(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(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 4.1 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 4.2. *Given any 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.*

4.2.2 The optimization problem

As discussed in Section 4.1, by Observation 4.1, $b(T^*)$ 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 4.2, 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 [30], 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.

Theorem 4.3. *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.*

Proof. We provide the details on searching $b(T^*)$ from $S(0) \cup S(1)$. We first search $S(0)$, which consists of interpoint distances of the points of $P(0)$.

An interval $(a_0, b_0]$, which is initialized to $(0, \infty]$, is maintained throughout the algorithm. Applying the distance selection algorithm on the points of $P(0)$, we can find the k-th smallest distance d of $S(0)$ in $O(n^{4/3} \log^2 n)$ time, with $k = 1/2 \cdot \binom{n}{2}$. Applying our decision algorithm of Theorem 4.2 with $\lambda = d$, we can decide whether $b(T^*) \leq d$ in $O(n^{4/3} \log^2 n)$ time. Depending on the result, we update the interval $(a_0, b_0]$ accordingly and choose an appropriate value k for the next iteration. In this way, after $O(\log n)$ iterations, we can obtain an interval $(a_0, b_0]$ containing $b(T^*)$ with $a_0, b_0 \in S(0)$ such that no value of $S(0)$ is in (a_0, b_0). The total running time is $O(n^{4/3} \log^3 n)$.

Following the same idea we search $S(1)$ using the point set $P(1)$, which will produce in $O(n^{4/3} \log^3 n)$ time an interval $(a_1, b_1]$ containing $b(T^*)$ with $a_1, b_1 \in S(1)$ such that no value of $S(1)$ is in (a_1, b_1).

It is not difficult to see that $b(T^*) = \min\{b_0, b_1\}$. Applying our decision algorithm with $\lambda = b(T^*)$ can produce an optimal moving spanning tree T^*. The total time of the algorithm is thus $O(n^{4/3} \log^3 n)$. □

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

In this section, we prove Theorem 4.1. 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

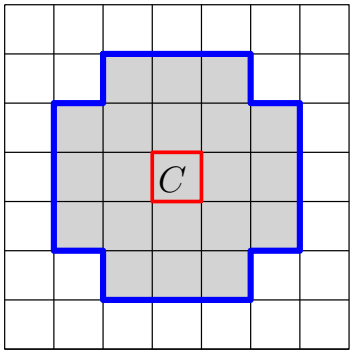


Fig. 4.3: The cells in the gray region bounded by the blue curve are all neighbors of the red cell.

distance between a point of C and a point of C' is at most λ (see Fig. 4.3). 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 4.2. *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 [1, 13, 17, 18]. The following lemma has been proved in [33].

Lemma 4.3. ([33])

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 4.3. We next prove Theorem 4.1 using the information computed in Lemma 4.3.

Consider a UDRE query with a query point p. By Lemma 4.3(2), we can determine whether p is in a cell $C \in \mathcal{C}$. If not, by Observation 4.2, 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 4.3.1, we first present some observations which our approach is based on. We describe our preprocessing algorithm for $Q(C')$ in Section 4.3.2 while handling the UDRE queries and deletions is discussed in Section 4.3.3. Section 4.3.4 finally summarizes everything. In the following, we let $m = |Q(C')|$.

4.3.1 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 [75] for maintaining the convex hull of a semi-dynamic (deletion-only) set of points in the plane.

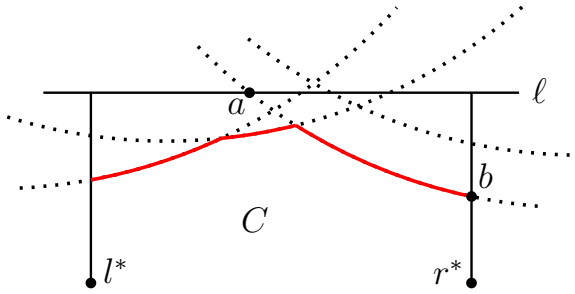


Fig. 4.4: Illustrating the lower envelope (the red curve).

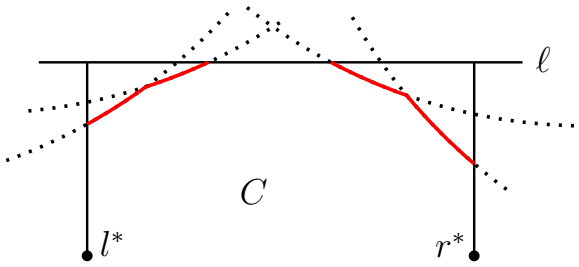


Fig. 4.5: Illustrating a lower envelope (the red curve) that has two connected components.

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. 4.4). 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. 4.4). 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. 4.4), 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.

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. 4.5). 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.

Lemma 4.4. *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').*

Proof. We prove the lemma by induction. Let $\xi_1, \xi_2, \ldots, \xi_m$ be the arcs of Ξ following the right-endpoint left-to-right order. For each $i = 1, 2, \ldots, m$, let $\Xi_i = \{\xi_1, \xi_2, \ldots, \xi_i\}$ and $\mathcal{U}_i$ denote the lower envelope of Ξ_i. We assume that the lemma statement holds for Ξ_{i-1} and $\mathcal{U}_{i-1}$, i.e., every arc of Ξ_{i-1} has at most one portion on $\mathcal{U}_{i-1}$ and the traversal order of $\mathcal{U}_{i-1}$ is consistent with the right-endpoint left-to-right order of Ξ_{i-1}, which is true when $i = 2$. Next we prove that the lemma statement holds for Ξ_i and $\mathcal{U}_i$. For each ξ_i, we use A_i to denote the unit-disk that has ξ_i on its boundary.

We add ξ_i to $\mathcal{U}_{i-1}$ since $\mathcal{U}_i$ is the lower envelope of ξ_i and $\mathcal{U}_{i-1}$. Let a and b be the left and right endpoints of ξ_i, respectively. Because the right endpoints of all arcs of Ξ_{i-1} are

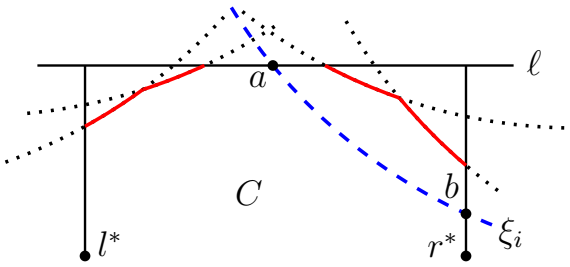


Fig. 4.6: Illustrating the first case in the proof of Lemma 4.4.

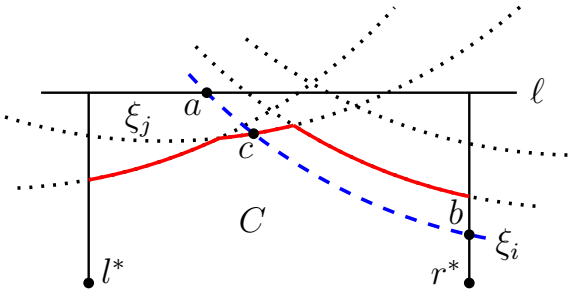


Fig. 4.7: Illustrating the second case in the proof of Lemma 4.4.

left of b, b must be on $\mathcal{U}_i$ and actually is the last point of $\mathcal{U}_i$ in the traversal order. Imagine that we move on ξ_i from b until we encounter either $\mathcal{U}_{i-1}$ or a, whichever first.

1. If we encounter a first, then ξ_i does not intersect $\mathcal{U}_{i-1}$ and thus the entire ξ_i is on $\mathcal{U}_i$ (see Fig. 4.6). Also, ξ_i is the last arc in the traversal order of $\mathcal{U}_i$ because b is the last point in the traversal. Therefore, the lemma statement holds for Ξ_i and $\mathcal{U}_i$. Note that it is possible that some components of $\mathcal{U}_{i-1}$ are covered by ξ_i, i.e., they are inside the disk A_i, in which case those components are not part of $\mathcal{U}_i$ anymore (see Fig. 4.6).

2. If we encounter $\mathcal{U}_{i-1}$ first, say, at a point c (see Fig. 4.7), then let ξ_j be the arc of Ξ_{i-1} that contains c. This means that ξ_i and ξ_j intersect at c. Due to our general position assumption, c is not an endpoint of either arc. As the two arcs have the same radius, ξ_i and ξ_j cross each other at c. Also, the portion of ξ_i between a and c is covered by ξ_j, i.e., they are inside the disk A_j, and thus cannot be on $\mathcal{U}_i$ (see Fig. 4.7). On the other hand, by the definition of c, the portion of ξ_i between c and b is part of $\mathcal{U}_i$ and is actually the last arc in the traversal order of $\mathcal{U}_i$ because b is the last point in the traversal. Therefore, the lemma statement holds for Ξ_i and $\mathcal{U}_i$. Note that the portion of $\mathcal{U}_{i-1}$ between c and its last point is covered by ξ_i, i.e., inside the disk A_i, and thus is not part of $\mathcal{U}_i$ anymore (see Fig. 4.7).

The above proves that the lemma holds for Ξ_i and $\mathcal{U}_i$. □

4.3.2 Preprocessing

We perform the following preprocessing algorithm for $Q(C')$. Due to Lemma 4.4, we are able to extend to our problem a technique from Hershberger and Suri [75] 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., [75, 76]): 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 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 4.4, is crucial to the success of our approach.

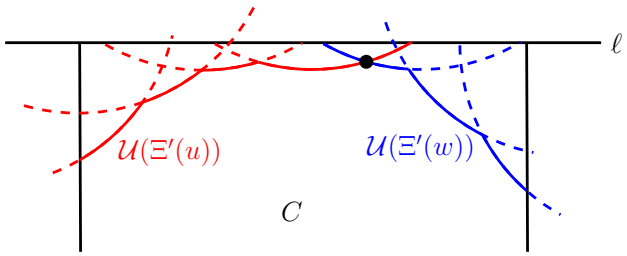


Fig. 4.8: Illustrating Lemma 4.5: The red (resp., blue) arcs are those from $\Xi'(u)$ (resp., $\Xi'(w)$). There is only one intersection between $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$.

Lemma 4.5. *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. 4.8).*

Proof. Note that $\mathcal{U}(\Xi'(v))$ is also the lower envelope of $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$. Assume to the contrary that $\mathcal{U}(\Xi'(u))$ and $\mathcal{U}(\Xi'(w))$ have two or more intersections. Then, $\mathcal{U}(\Xi'(v))$ has three arcs ξ_1, ξ_2, and ξ_3 following the traversal order such that both ξ_1 and ξ_3 are from one of the two subsets $\Xi'(u)$ and $\Xi'(w)$ while ξ_2 is from the other. This implies that the traversal order of $\mathcal{U}(\Xi'(v))$ is not consistent with the right-endpoint left-to-right order of $\Xi'(v)$ because right endpoints of all arcs of $\Xi'(u)$ are left of the right endpoints of all arcs of $\Xi'(w)$, a contradiction to Lemma 4.4. □

By Lemma 4.5, 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 4.3.3, 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 4.6. *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 4.5, $\mathcal{U}(\Xi(u))$ 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.

4.3.3 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.

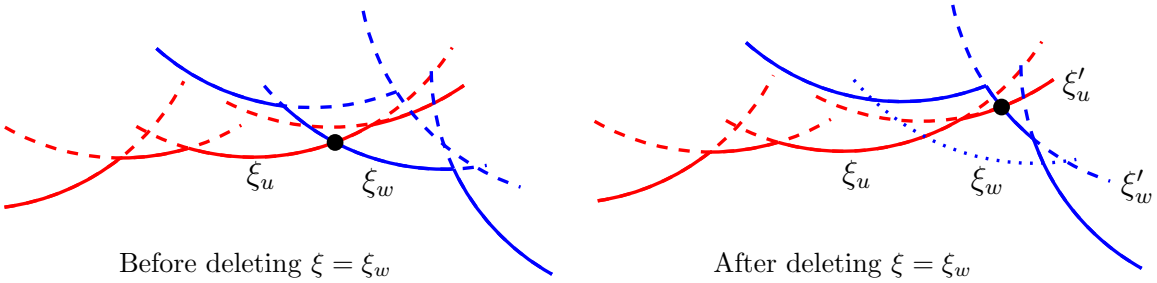
Lemma 4.7. *Deleting an arc from the envelope tree $T(\Xi')$ can be done in $O(\log m)$ amortized time.*

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



Before deleting $\xi = \xi_w$ After deleting $\xi = \xi_w$

Fig. 4.9: Illustrating the deletion of $\xi = \xi_w$. The red (resp., blue) arcs are those from $\Xi'(u)$ (resp., $\Xi'(w)$).

construction algorithm in Lemma 4.6, 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. 4.9). 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. 4.9). As such, we compute ξ'_u and ξ'_w using a line sweep procedure that is similar to the algorithm in Lemma 4.6, 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 4.6.

The pseudocode in Algorithm 4.1 summarizes the algorithm.

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

Algorithm 4.1: Deleting an arc ξ from the envelope tree $T(\Xi')$.

```
1  Function Delete(ξ, v):
       // Initially, v is the root of the envelope tree T(Ξ').
       // Let z be the leaf that stores ξ. We assume that z is in the
          right subtree of v; the other case is symmetric.
       // In the beginning of this procedure, arcs(v) stores U(Ξ'(v)); at
          the end arcs(v) stores U(Ξ'(v) \ {ξ}).
2      u = v.left_child
3      w = v.right_child
4      Restore U(Ξ'(u)) and U(Ξ'(w)) using arcs(u), arcs(w), arcs(v), and X(v).
5      if w = z then
6          v.right_child = NULL
7          arcs(v) = arcs(u)
8          X(v) = NULL
9          arcs(u) = NULL
10     end
11     else
12         if ξ ∈ X(v) = {ξ_u, ξ_w} then
13             ξ_1 = ξ_u and ξ_2 is set to the left neighbor of ξ_w in arcs(w)
14             Delete(ξ, w)
15             Using the line sweep procedure of Lemma 4.6 (but starting from the left
                arc of ξ_1 and ξ_2) to find the intersection of arcs(u) and arcs(w), and
                then set X(v).
16             Cut off arcs(u) and arcs(w) at the intersection and concatenate the
                corresponding parts to produce arcs(v) (similar to the algorithm of
                Lemma 4.6)
17         end
18         else
19             Delete(ξ, w)
20             Cut off arcs(u) and arcs(w) at the intersection and concatenate the
                corresponding parts to produce arcs(v) (similar to the algorithm of
                Lemma 4.6).
21         end
22     end
23 end
```

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. 4.9). 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. □

4.3.4 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 4.1, 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 4.3(2) to determine whether p is in a cell of $\mathcal{C}$. If not, then by Observation 4.2, $A_p \cap Q = \emptyset$ and thus we are done with the query. Otherwise, Lemma 4.3(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 4.3(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 4.1.

4.4 Concluding remarks

In this chapter, 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 [20]. 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 [30] 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 4.1 holds.

CHAPTER 5

A SIMPLE ALGORITHM FOR UNIT-DISK RANGE REPORTING

5.1 Introduction

We design a simple but optimal algorithm for unit-disk range reporting. The result in this chapter has been submitted to a conference and is now still under review.

5.1.1 Problem definitions and our results

Given a set P of n points in the plane and a value r, we consider the following *unit-disk range reporting* problem (or UDRR for short): Construct a data structure such that given any query disk of radius r, all points of P in the disk can be reported efficiently. Without loss of generality, we assume $r = 1$ and thus each query disk is a *unit disk*.

The UDRR problem is also known as the *fixed-radius neighbor problem* in the literature [22–25]. Chazelle and Edelsbrunner [25] constructed a data structure of $O(n)$ space that can answer each query in $O(\log n + k)$ time, where k is the output size; their data structure can be constructed in $O(n^2)$ time. By a standard lifting transformation [80], the problem can be reduced to the half-space range reporting queries in 3D; this reduction also works if the radius of the query disk is arbitrary. Using Afshani and Chan's 3D half-space range reporting data structure [26], one can construct a data structure of $O(n)$ space with $O(k + \log n)$ query time, while the preprocessing takes $O(n \log n)$ expected time since it invokes Ramos' algorithm [81] to construct shallow cuttings for a set of planes in 3D, which is the only randomized part of the whole algorithm. Chan and Tsakalidis [27] later presented an $O(n \log n)$-time deterministic algorithm for the shallow cutting problem. Therefore, combining the framework in [26] with the shallow cutting algorithm [27], one can build a data structure of $O(n)$ space in $O(n \log n)$ deterministic time that can answer each UDRR query in $O(k + \log n)$ time; note that this result also works for query disks of

arbitrary radii.

By exploiting special properties of unit disks, we present a new UDRR data structure with the same complexity as above. Our algorithm is much simpler than the algorithm of [26, 27]. Indeed, the algorithm of [26, 27] involves relatively advanced geometric techniques like shallow partition theorem and shallow cuttings in 3D, planar graph separators, computing ϵ-net and ϵ-approximations, etc. Our algorithm, in contrast, only relies on elementary techniques (the most complicated one might be a fractional cascading data structure [28, 29]). One may consider our algorithm a generalization of the classical 2D half-plane range reporting algorithm of Chazelle, Guibas, and Lee [82].

A closely related problem is the *unit-disk range counting problem*, which is to compute the number of points of P in the query disk. By the standard lifting transformation, the problem can be reduced to 3D half-space range counting [77–79]. One can also solve the problem using the algorithms for general semialgebraic range counting, e.g., [83, 84]. Wang [33] recently presented various algorithms by extending the classical techniques for 2D half-plane range counting [77–79]. Refer to [85–87] for surveys on range searching in general and some recent progress on semialgebraic range reporting [88, 89].

Our approach. We first build a grid to capture the proximity information for points of P. The side length of each grid cell is $1/\sqrt{2}$ so that the distance between any two points in the same grid cell is at most 1. For any query unit disk D_q whose center is q, points of P in the grid cell C that contains q can be reported immediately. The critical part is to handle other cells containing points of $P \cap D_q$. The number of such cells is constant and each of them is separated from C (and thus from q) by an axis-parallel line. The problem thus boils down to the following subproblem: Given a set Q of points in a grid cell C' above a horizontal line ℓ, report points of Q in any query unit disk whose center is below ℓ. A point $p \in Q$ is in D_q if and only if q lies in the unit disk D_p centered at p, or equivalently, q is above the arc of the boundary of D_p below ℓ. Let $\mathcal{A}$ denote the set of all such arcs for all points $p \in Q$. To find the points of Q in D_q, it suffices to determine the arcs of $\mathcal{A}$ below q. To tackle this problem, we follow the same framework as that for the 2D half-plane range reporting algorithm [82].

More specifically, in the preprocessing, we construct layers of lower envelopes of $\mathcal{A}$ and then build a fractional cascading data structure on them [28, 29]. Using the fractional cascading data structure, the arcs of $\mathcal{A}$ below each query point q can be reported in $O(k + \log |Q|)$ time, where k is the output size. The success of our method hinges on computing the layers of lower envelopes of $\mathcal{A}$ in $O(|Q| \log |Q|)$ time. To this end, we consider its dual problem that is to compute the layers of the α-hull [90] of Q with $\alpha = -1$. By generalizing Chazelle's algorithm [91] for computing convex hull layers, we show that the α-hull layers of Q can be computed in $O(|Q| \log |Q|)$ time. Our algorithm is actually simpler than Chazelle's original algorithm [91] since we do not need to handle cross deletions in our problem.

Outline. We present our algorithm for the UDRR problem in Section 5.2. The algorithm uses a subroutine that computes layers of lower envelopes of circular arcs as discussed above; the subroutine is described in Section 5.3. Section 5.4 concludes the chapter as well as demonstrates that our techniques may be used to solve other related problems (e.g., *outside-unit-disk range reporting* and *unit-disk range emptiness* queries).

5.2 The UDRR algorithm

Let P be a set of n points in the plane. We wish to construct a data structure on P so that given any query unit disk, the points of P in the query disk can be reported efficiently.

For any point q, let D_q denote the unit disk centered at q. For any region R and any set Q of points in the plane, let $Q(R)$ denote the subset of points of Q inside R, i.e., $Q(R) = Q \cap R$. Unless otherwise stated, a circular arc or an arc refers to a circular arc of radius 1. For a circular arc A, we call the disk whose boundary contains A the *underlying disk* of A.

For any region R in the plane, we use ∂R to denote its boundary. For any point p in the plane, we use $x(p)$ to denote the x-coordinate of p.

5.2.1 Constructing a grid

Our preprocessing algorithm starts with implicitly building a grid Ψ, which partitions

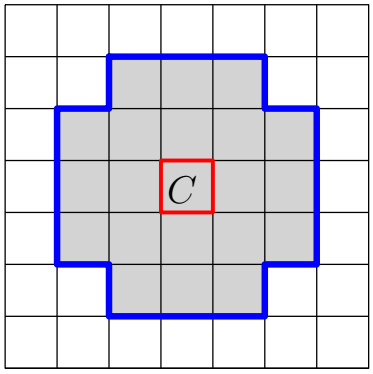


Fig. 5.1: Illustrating the grid Ψ, where P consists of the three black points and $\mathcal{C}$ consists of all gray square cells. Any point whose distance to q is at most 1 must lie in the region bounded by blue segments, which contains 21 cells (those cells constitute $N(C)$).

the plane into square cells of side length $1/\sqrt{2}$ by axis-parallel lines. This guarantees that the distance of any two points in each cell of Ψ is at most 1. For ease of exposition, we assume that every point of P lies in the interior of a cell of Ψ. We say that a cell C' is a *neighbor* of another cell C if the minimum distance between points of C and points of C' is at most 1. We use $N(C)$ to denote the set of all neighbors of cell C in Ψ. We also let $N(C)$ contain cell C itself. Note that $|N(C)| = O(1)$ and $C' \in N(C)$ if and only if $C \in N(C')$. Observe that for any point q in a cell C, any point whose distance to q is at most 1 must lie in a cell of $N(C)$. Define $\mathcal{C}$ to be the set of cells of Ψ that contain at least one point of P along with their neighbors, i.e., $\mathcal{C} = \bigcup_{C \cap P \neq \emptyset} N(C)$; see Fig. 5.1. Note that $\mathcal{C}$ has $O(n)$ cells. The following observation follows the definition of $\mathcal{C}$.

Observation 5.1. *For any point q in the plane, if q is not in any cell of $\mathcal{C}$, then $P \cap D_q = \emptyset$.*

The technique of using grids has been widely used in various algorithms for solving problems in unit-disk graphs [1, 13, 14, 17, 18, 21, 33]. The following lemma has been proved in [33].

Lemma 5.1. ([33])

1. *The set $\mathcal{C}$, along with the sets $P(C)$ 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 q, we can do the following in $O(\log n)$ time: Determine whether q is in a cell C of $\mathcal{C}$, and if yes, return C and the set $N(C)$.*

Note that we only need to compute the information in Lemma 5.1 rather than the entire grid Ψ. With Lemma 5.1(2) and Observation 5.1, for any query unit disk D_q whose center is q, if q is not in a cell of $\mathcal{C}$, then $P(D_q) = \emptyset$ and thus we simply return null. In the following, we assume that q lies in a cell $C \in \mathcal{C}$. Our goal is to report $P(D_q)$. To this end, it suffices to report $P(C') \cap D_q$ for all cells $C' \in N(C)$. In the case of $C' = C$, since the distance between any two points in C is at most 1, we can simply report all points of $P(C)$. In what follows, we focus on the case $C' \neq C$.

Since $C' \neq C$, C and C' are separated by an axis-parallel line. Without loss of generality, we assume that C and C' are separated by a horizontal line ℓ with C' above ℓ and C below ℓ. As $q \in C$, q is below ℓ, i.e., q is separated from C' by ℓ. Our target is to report points of $P(C') \cap D_q$. We formulate the problem as the following subproblem, called the *line-separable UDRR problem*:

Problem 5.1. *(Line-separable UDRR) Given a set Q of m points above a horizontal line ℓ such that all points of Q are contained in a unit disk, build a data structure so that for any query unit disk D_q centered at a point q below ℓ, the points of Q in D_q can be reported efficiently.*

For solving our problem, we can set $Q = P(C')$ since all points of $P(C')$ are in C', which is contained in a unit disk. In what follows, we will prove Lemma 5.2.

Lemma 5.2. *For the line-separable UDRR, we can build a data structure of $O(m)$ space in $O(m \log m)$ time that can answer each query in $O(k + \log m)$ time, where k is the output size.*

Before proving Lemma 5.2, we prove the following main result for UDRR using Lemma 5.2.

Theorem 5.1. *Given a set P of n points in the plane, we can build a data structure of $O(n)$ space in $O(n \log n)$ time such that given any query unit disk, the points of P in the disk can be reported in $O(\log n + k)$ time, where k is the output size.*

Proof. We first compute the information in Lemma 5.1. Then, for each cell $C \in \mathcal{C}$, for each $C' \in N(C)$, we construct the line-separable UDRR data structure of Lemma 5.2 for $P(C')$ with respect to C (i.e., by rotating the plane so that C' and C are separated by a horizontal line and C' is above the line), which takes $O(|P(C')|)$ space and $O(|P(C')| \cdot \log |P(C')|)$ time. This finishes the preprocessing. Since $\bigcup_{C' \in \mathcal{C}} P(C') = P$ and each cell C' belongs to $N(C)$ for a constant number of cells $C \in \mathcal{C}$, the preprocessing takes $O(n)$ space and $O(n \log n)$ time in total.

Given a query unit disk D_q centered at a point q, we first check whether q is in a cell of $\mathcal{C}$ and if yes find such a cell, which can be done in $O(\log n)$ time by Lemma 5.1(2). If no cell of $\mathcal{C}$ contains q, then by Observation 5.1 we can simply return null. Otherwise, let C be the cell of $\mathcal{C}$ that contains q. We first report all points of $P(C)$. Next, for each $C' \in N(C)$, using the line-separable UDRR data structure we built for $P(C')$ with respect to C, we report all points of $P(C')$ inside D_q. As $|N(C)| = O(1)$, the total query time is $O(k + \log n)$ by Lemma 5.2. □

5.2.2 Line-separable UDRR: Proving Lemma 5.2

We now prove Lemma 5.2.

Consider a query unit disk D_q whose center q is below ℓ. The goal of the query is to report $Q \cap D_q$. Observe that a point $p \in Q$ is in D_q if and only if q is in the unit disk D_p. The portion of ∂D_p below ℓ is a circular arc, denoted by A_p. Since p is above ℓ, A_p is on the lower half circle of ∂D_p and thus is x-monotone. As such, p is in D_q if and only if q is above the arc A_p. Since all our query disk centers are below ℓ, if $A_p = \emptyset$, then p can be ignored since it is either in all query disks or not in any query disk. Without loss of generality, we assume that $A_p \neq \emptyset$ for all $p \in Q$. Define $\mathcal{A}$ to be the set of arcs A_p for all points $p \in Q$. As such, reporting the points of Q in D_q becomes reporting the arcs of $\mathcal{A}$ that are below q.

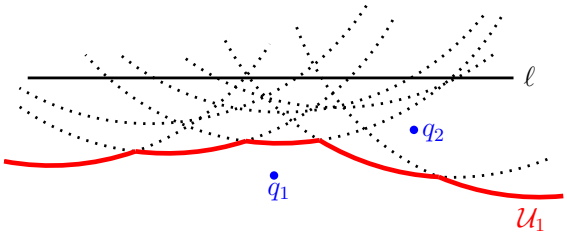


Fig. 5.2: Illustrating the lower envelope $\mathcal{U}_1$. Black dotted arcs are boundaries of unit disks centered at points of Q. The point q_1 is below $\mathcal{U}_1$ while q_2 is above $\mathcal{U}_1$.

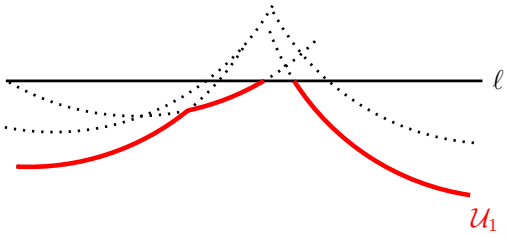


Fig. 5.3: Illustrating a lower envelope $\mathcal{U}_1$ with two connected components.

Define $\mathcal{U}_1$ as the lower envelope of the arcs of $\mathcal{A}$ (see Fig. 5.2). Since each arc of $\mathcal{A}$ is x-monotone, $\mathcal{U}_1$ is also x-monotone. Note that $\mathcal{U}_1$ may have several connected components (see Fig. 5.3). Observe that q is above an arc of $\mathcal{A}$ if and only if q is above $\mathcal{U}_1$ (see Fig. 5.2). It has been proved by Wang and Zhao [21] that each arc of $\mathcal{A}$ can contribute at most one arc in $\mathcal{U}_1$. Suppose we traverse arcs of $\mathcal{U}_1$ from left to right; the order of these arcs encountered during our traversal is called the *traversal order*. The following lemma shows that the traversal order is consistent with the order of the arcs of $\mathcal{U}_1$ sorted by their centers from left to right.

Lemma 5.3. *The centers of arcs in $\mathcal{U}_1$ following the traversal order are sorted in ascending order by x-coordinate.*

Proof. Consider two consecutive arcs $\mathcal{U}_1$ following the traversal order and let A_i and A_{i+1} be the two arcs of $\mathcal{A}$ containing them, respectively. Let p_i and p_{i+1} are the centers of A_i and A_{i+1}, respectively. Our goal is to prove that $x(p_i) \leq x(p_{i+1})$.

Let a_j and b_j be the left and right endpoints of A_j, for $j \in \{i, i+1\}$. It has been proved in [21] that $x(b_i) \leq x(b_{i+1})$ because the subarc of A_i on $\mathcal{U}_1$ appears in the front of that of A_{i+1} in the traversal order. There are two cases depending on whether A_i and A_{i+1} intersect. If they do not intersect, then since $x(b_i) \leq x(b_{i+1})$, it holds that $x(b_i) \leq x(a_{i+1})$. As such, $x(p_i) \leq x(p_{i+1})$ must hold since $x(p_j) = (x(a_j) + x(b_j))/2$ for $j \in \{i, i+1\}$. If A_i and A_{i+1} intersect, then they intersect exactly once since their underlying disks are unit

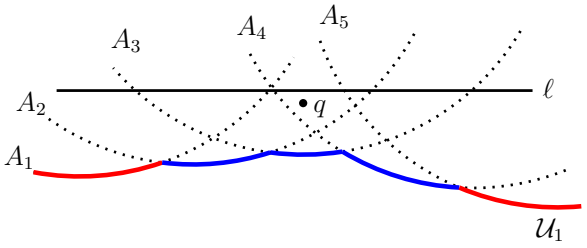


Fig. 5.4: The three blue arcs are below q while the two red arcs are above q.

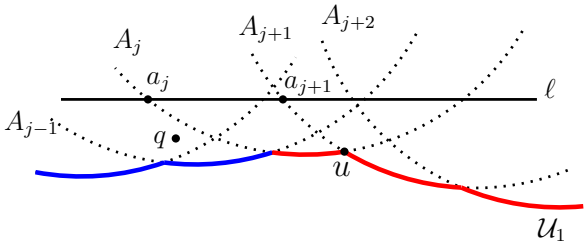


Fig. 5.5: Illustrating the case where A'_j and A'_{j+1} intersect at a vertex u of $\mathcal{U}_1$.

disks. As such, since $x(b_i) \leq x(b_{i+1})$, we have $x(a_i) \leq x(a_{i+1})$. Therefore, $x(p_i) \leq x(p_{i+1})$ must hold since $x(p_j) = (x(a_j) + x(b_j))/2$ for $j \in \{i, i+1\}$. □

Define Q_1 to be the set of centers of all arcs of $\mathcal{U}_1$. We say that an arc A of $\mathcal{U}_1$ *spans* a point p, if $x(p)$ is between the x-coordinates of the two endpoints of A.

Lemma 5.4. *Suppose q is a point below ℓ and the arc of $\mathcal{U}_1$ spanning q is known; then the points of $Q_1 \cap D_q$ can be reported in $O(|Q_1 \cap D_q|)$ time (assuming that $\mathcal{U}_1$ is stored in a data structure so that one can access from each arc of $\mathcal{U}_1$ its neighboring arcs in $O(1)$ time).*

Proof. Let $A'_1, A'_2, \ldots, A'_t$ be the arcs of $\mathcal{U}_1$ following their traversal order, where t is the number of arcs of $\mathcal{U}_1$. For each $1 \leq i \leq t$, let p_i be the center of A'_i and A_i be the arc of $\mathcal{A}$ containing A'_i. By definition, $Q_1 = \{p_1, p_2, \ldots, p_t\}$.

If no arc of $\mathcal{U}_1$ spans q, then it is not difficult to see that $Q_1 \cap D_q = \emptyset$. In the following, we assume that $\mathcal{U}_1$ has an arc spanning q, denoted by A'_i.

If q is below A'_i, then q is below $\mathcal{U}_1$ and thus $Q_1 \cap D_q = \emptyset$. We thus assume that q is above A'_i (see Fig. 5.4, where A'_i is A'_3). In this case, p_i is in D_q and we report it. Next, starting from A'_i, we traverse on the arcs of $\mathcal{U}_1$ rightwards (resp., leftwards) until the distance between q and the center of an arc is larger than 1. Specifically, for the rightwards case, we check the arcs of $\{A'_{i+1}, A'_{i+2}, ...\}$ in this order and for each arc A'_j, $j \geq i+1$, if p_j is in D_q, then we report p_j and proceed on $j+1$; otherwise, we halt the procedure. The leftwards case is symmetric. To see the correctness, we only argue the rightwards case as the other case is symmetric.

Suppose p_j is outside D_q. Our goal is to show that p_h is not in D_q for any $j+1 \leq h \leq t$. Consider the arc A'_{j+1}. There are two cases depending on whether A'_j and A'_{j+1} intersect. Let a_i and b_i be the left and right endpoints of A_i, respectively, for $i \in \{j, j+1\}$.

- If A'_j and A'_{j+1} intersect, say, at a point u, then u is a vertex of $\mathcal{U}_1$ (see Fig. 5.5). As q is spanned by A'_i and $i < j$, it holds that $x(q) < x(u)$. Since p_j is outside D_q, q is not above A_j, and more specifically, not above the portion of A_j between a_j and u. Since A'_j and A'_{j+1} intersect and both arcs have the same radius, the portion of A_j between a_j and u is below the portion of A_{j+1} between a_{j+1} and u. Since q is not above the portion of A_j between a_j and u, q cannot be above the portion of A_{j+1} between a_{j+1} and u. As $x(q) < x(u)$, this implies that q cannot be above A_{j+1} and thus p_{j+1} cannot be in D_q.

- If A'_j and A'_{j+1} do not intersect, then both the right endpoint b_j of A_j and the left endpoint a_{j+1} of A_{j+1} are vertices of $\mathcal{U}_1$ and $x(b_j) < x(a_{j+1})$. As q is spanned by A'_i and $i < j$, $x(q) \leq x(b_j)$, and thus $x(q) < x(a_{j+1})$. Hence, q cannot be above A_{j+1} and therefore p_{j+1} cannot be in D_q.

The above proves that p_{j+1} cannot be in D_q. Following the same analysis, we can show that p_h cannot be in D_q for all $h = j+2, j+3, \ldots, t$.

Clearly, the algorithm runs in $O(k)$ time, where $k = |Q_1 \cap D_q|$. This proves the lemma. □

By Lemma 5.4, if we store arcs of $\mathcal{U}_1$ by a balanced binary search tree, given a query point q below ℓ, the arc of $\mathcal{U}_1$ spanning q can be computed in $O(\log m)$ time and consequently $Q_1 \cap D_q$ can be reported in additional $O(|Q_1 \cap D_q|)$ time. Recall that our goal is to report $Q \cap D_q$. To report the remaining points, i.e., those of $Q \setminus Q_1$ in D_q, we apply the idea recursively on $Q \setminus Q_1$. Specifically, define $\mathcal{U}_2$ as the lower envelope of the arcs of $\mathcal{A}$ after the arcs defined by the points of Q_1 are removed; let Q_2 denote the set of centers of the arcs of $\mathcal{U}_2$. In general, define $\mathcal{U}_i$ as the lower envelope of the arcs of $\mathcal{A}$ after the arcs defined by the points of $\bigcup_{j=1}^{i-1} Q_j$ are removed for $i = 2, 3, \ldots$ (see Fig. 5.6); let Q_i denote the set

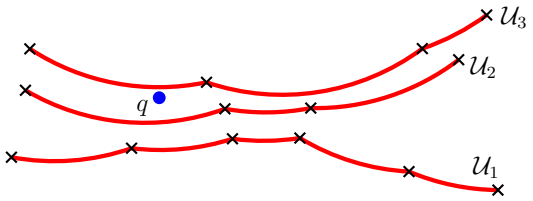


Fig. 5.6: Illustrating layers of lower envelopes $\mathcal{U}_1$,$\mathcal{U}_2$,$\mathcal{U}_3$.

of centers of the arcs of $\mathcal{U}_i$. We call $\{\mathcal{U}_i\}$ the *lower envelope layers* of $\mathcal{A}$. The following theorem, which will be proved in Section 5.3, computes the lower envelope layers.

Theorem 5.2. *The lower envelope layers of $\mathcal{A}$ can be computed in $O(m \log m)$ time and $O(m)$ space, where $m = |\mathcal{A}|$.*

Proving Lemma 5.2. We now have all ingredients to prove Lemma 5.2. We compute the lower envelope layers of $\mathcal{A}$ by Theorem 5.2. Then, we construct a fractional cascading data structure on the vertices of the lower envelope layers [28,29]. This finishes the preprocessing, which takes $O(m)$ space and $O(m \log m)$ time in total. Given a query unit disk D_q centered at a point q below the line ℓ, using the fractional cascading data structure, we can compute the arc of $\mathcal{U}_1$ spanning q in $O(\log m)$ time and compute the arc of the next layer $\mathcal{U}_2$,$\mathcal{U}_3$,... spanning q in $O(1)$ time each. We compute the arc A'_i of $\mathcal{U}_i$ that spans q for all $i = 1, 2, \ldots$ until an index j such that q is below A'_j (and thus Q_j does not have any point in D_q, which is also the case for $Q_{j+1}, Q_{j+2}, \cdots$). Then, for each $\mathcal{U}_i$ with $1 \leq i \leq j-1$, using the arc A'_i, we apply Lemma 5.4 to report the points of $Q_i \cap D_q$. Because q is above $\mathcal{U}_i$ for each $1 \leq i \leq j-1$, Q_i has at least one point in D_q. As such, the total time of the query algorithm is bounded by $O(k + \log m)$, where $k = |Q \cap D_q|$. This proves Lemma 5.2.

5.3 Computing layers of lower envelopes

In this section, we prove Theorem 5.2. We follow the same notation as before, e.g., Q, $\mathcal{A}$, $\mathcal{U}_i$, Q_i, except that we now use n to denote $|Q|$ for convenience. Recall that all points of Q are contained in a unit disk and thus the distance of every two points of Q is at most

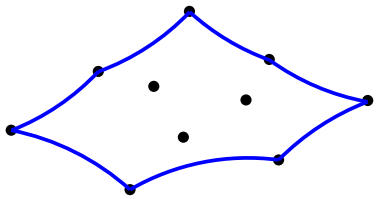


Fig. 5.7: Illustrating the α-hull of Q, for $\alpha = -1$.

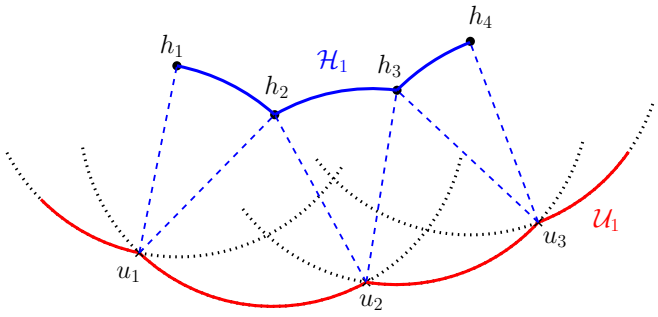


Fig. 5.8: Illustrating the lower α-hull $\mathcal{H}_1$ of Q and the lower envelope $\mathcal{U}_1$ of $\mathcal{A}$. Black dotted arcs are boundaries of underlying disks of arcs of $\mathcal{U}_1$. Vertices of $\mathcal{H}_1$ are centers of arcs of $\mathcal{U}_1$, and vice versa.

1. For ease of exposition, we assume that no two points of Q have the same x-coordinate. For any subset $Q' \subseteq Q$, define $\mathcal{A}(Q') = \{A_p \mid p \in Q'\}$.

Our goal is to compute the lower envelope layers $\{\mathcal{U}_i\}$. Instead of computing them directly, we consider a *dual problem*. We borrow a concept *α-hull* from [90], which is a generalization of the convex hull. For a real number α, a *generalized disk* of radius $1/\alpha$ is defined to be a disk of radius $1/\alpha$ if $\alpha > 0$, the complement of a disk of radius $-1/\alpha$ if $\alpha < 0$, and a halfplane if $\alpha = 0$. The *α-hull* of Q is the intersection of all generalized disks with radius $1/\alpha$ that contain all points of Q (see Fig. 5.7). For our problem, we are interested in the case $\alpha = -1$. Henceforth, unless other stated, $\alpha = -1$.

It is known that the leftmost (resp., rightmost) point of Q must be the leftmost (resp., rightmost) vertex of the α-hull of Q [90]. The *lower α-hull* of Q, denoted by $\mathcal{H}_1$, is defined as the portion of the boundary of the α-hull counterclockwise from its leftmost vertex to its rightmost vertex (similar concepts have been used elsewhere, e.g., [92]).

For any two points p and p' of Q, as their distance is at most 1, there are two circular arcs of radius 1 connecting them. One of these arcs having its center below the line through p and p' while the other having its center above the line (recall that $x(p) \neq x(p')$ due to our assumption); we call the former arc the *concave arc* of p and p', denoted by $A(p, p')$. Note that the lower α-hull $\mathcal{H}_1$ comprises of concave arcs [90].

We observe the following *duality* between the lower hull $\mathcal{H}_1$ of Q and the lower envelope

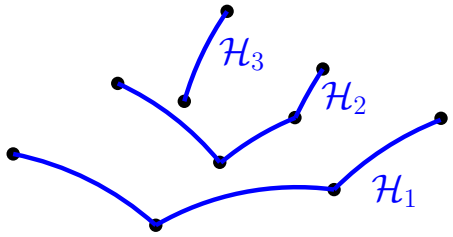


Fig. 5.9: Illustrating lower α-hull layers $\{\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3\}$.

$\mathcal{U}_1$ of $\mathcal{A}$ (see Fig. 5.8): The center of each arc in $\mathcal{H}_1$ is a vertex of $\mathcal{U}_1$ while the center of each arc of $\mathcal{U}_1$ is a vertex of $\mathcal{H}_1$. Due to this duality, Q_1 is exactly the set of vertices of $\mathcal{H}_1$.

Like lower envelope layers of $\mathcal{A}$, we can correspondingly define *lower α-hull layers* of Q. Specifically, define $\mathcal{H}_2$ as the lower α-hull of $Q \setminus Q_1$, i.e., the remaining points of Q after vertices of $\mathcal{H}_1$ are removed; $\mathcal{H}_i$ is defined similarly for $i = 3, 4, \ldots$; see Fig. 5.9. As above, each $\mathcal{H}_i$ is dual to $\mathcal{U}_i$, and thus Q_i is the set of vertices of $\mathcal{H}_i$. As such, to compute layers of lower envelopes $\{\mathcal{U}_i\}$ of $\mathcal{A}$, it suffices to compute layers of lower α-hulls $\{\mathcal{H}_i\}$ of Q, which is our focus below.

We present an algorithm to compute the lower α-hull layers $\{\mathcal{H}_i\}$ in $O(n)$ space and $O(n \log n)$ time. We follow the scheme of Chazelle's algorithm [91] for computing convex hull layers of a set of points in the plane. Our algorithm is actually simpler since cross deletions are not needed in our algorithm. The main idea is to construct a *tree graph* G embedded in the plane such that each edge is a circular arc. $\mathcal{H}_1$ can be produced in $O(|\mathcal{H}_1|)$ time by using G. Then, vertices of $\mathcal{H}_1$ are removed from G and G is updated so that $\mathcal{H}_2$ can be produced in $O(|\mathcal{H}_2|)$ time. Repeating this process until G becomes $\emptyset$ will produce the lower α-hull layers $\{\mathcal{H}_i\}$. In what follows, we first define the graph G in Section 5.3.1 and then describe an algorithm to construct it in Section 5.3.2. Finally in Section 5.3.3 we compute lower α-hull layers using G.

5.3.1 Defining the tree graph G

Let $p_1, p_2, ..., p_n$ be the list of the points of Q sorted from left to right. Let T be a complete binary tree whose leaves store $p_1, p_2, ..., p_n$ from left to right, respectively. For each node v of T, let $Q(v) \subseteq Q$ be the set of points that are stored at the leaves of the

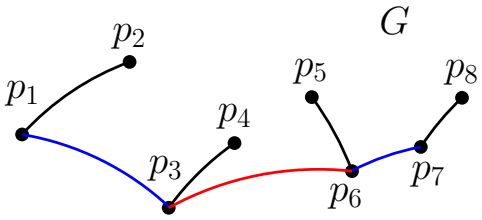


Fig. 5.10: Illustrating the graph G for a set $Q = \{p_1, p_2, ..., p_8\}$ of 8 points.

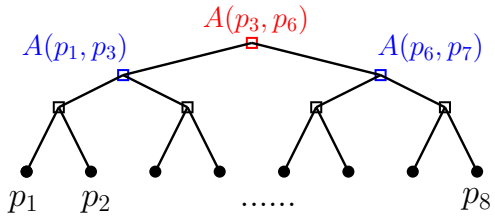


Fig. 5.11: Illustrating T for the example in Fig. 5.10. Internal nodes store common tangent arcs, which are edges of G.

subtree rooted at v and let $\mathcal{A}(v) = \mathcal{A}(Q(v))$. Let $\mathcal{H}(v)$ denote the lower α-hull of points in $Q(v)$ and $\mathcal{U}(v)$ the lower envelope of $\mathcal{A}(v)$. Hence, $\mathcal{H}(v)$ and $\mathcal{U}(v)$ are dual to each other.

The graph G is defined as follows: Its vertex set is Q and its edge set consists of arcs of $\mathcal{H}(v)$ of all nodes v of T (see Fig. 5.10). As such, each edge of G is a concave arc.

For any vertex p of the lower α-hull $\mathcal{H}$ of a subset Q' of Q, we say that a circular arc A containing p is *tangent* to $\mathcal{H}$ at p if no point of Q' is contained in the interior of the underlying disk of A. Note that A is tangent to $\mathcal{H}$ if and only if the two adjacent vertices of p on $\mathcal{H}$ are outside the underlying disk of A.

Consider a node $v \in T$. Let u and w be v's left and right children, respectively. A concave arc $A(p_i, p_j)$ connecting a vertex p_i of $\mathcal{H}(u)$ and a vertex p_j of $\mathcal{H}(w)$ is called a *common tangent arc* of $\mathcal{H}(u)$ and $\mathcal{H}(w)$ if $A(p_i, p_j)$ is tangent to $\mathcal{H}(u)$ at p_i and tangent to $\mathcal{H}(w)$ at p_j. By duality, $A(p_i, p_j)$ corresponds to the intersection a between $\mathcal{U}(u)$ and $\mathcal{U}(w)$ (i.e., a is the center of $A(p_i, p_j)$). It has been proved in [21] that $\mathcal{U}(u)$ and $\mathcal{U}(w)$ have at most one intersection, and thus $\mathcal{H}(u)$ and $\mathcal{H}(w)$ have at most one common arc tangent. In fact, since all points of Q are contained in a unit disk, $\mathcal{H}(u)$ and $\mathcal{H}(w)$ have exactly one common tangent arc, say, $A(p_i, p_j)$, connecting a vertex p_i of $\mathcal{H}(u)$ and a vertex p_j of $\mathcal{H}(w)$. Then $\mathcal{H}(v)$ consists of the following three portions in order from left to right: the portion of $\mathcal{H}(u)$ between its leftmost vertex and p_i, the arc $A(p_i, p_j)$, and the portion of $\mathcal{H}(w)$ between p_j and its rightmost vertex. We store $A(p_i, p_j)$ at v, denoted by $A(v)$; see Fig. 5.11. The common tangent arcs $A(v)$ for all internal nodes v of T form exactly the edge set of G.

We store the graph G in an adjacency-list structure as follows. Each vertex p of G is associated with two doubly linked lists $L_l(p)$ and $L_r(p)$ such that $L_l(p) \cup L_r(p)$ contains all

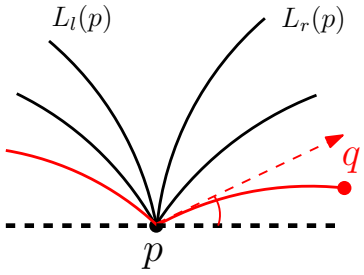


Fig. 5.12: Illustrating the adjacency lists $L_l(p)$ and $L_r(p)$ at p. The two red arcs are bottom edges. The red dashed segment with arrow is the tangent ray of $A(p, q)$ at p and the tangent angle is shown.

adjacent vertices of p in G, where $L_l(p)$ (resp., $L_r(p)$) stores adjacent vertices of p that are to the left (resp., right) of p. For each adjacent vertex q of p, we define the *tangent angle* of the concave arc $A(p, q)$ of G connecting p and q as the acute angle of the tangent ray of $A(p, q)$ at p following the direction toward q with the horizontal line through p (see Fig. 5.12). Vertices of $L_l(p)$ (resp., $L_r(p)$) are sorted by the tangent angles of their corresponding arcs. The *bottom edge* of $L_l(p)$ (resp., $L_r(p)$) is defined as the arc with the minimum tangent angle in $L_l(p)$ (resp., $L_r(p)$); see Fig. 5.12. We add two pointers at p to access the two bottom edges in $L_l(p)$ and $L_r(p)$.

5.3.2 Constructing the tree graph G

The following lemma will be used as a subroutine in our algorithm for constructing G.

Lemma 5.5. *Given the lower α-hull $\mathcal{H}'$ of a subset $Q' \subseteq Q$ and the lower α-hull $\mathcal{H}''$ of another subset $Q'' \cap Q$ such that Q' and Q'' are separated by a vertical line, the common tangent arc of $\mathcal{H}'$ and $\mathcal{H}''$ can be computed in $O(|\mathcal{H}'| + |\mathcal{H}''|)$ time.*

Proof. Without loss of generality, we assume that $\mathcal{H}'$ is to the left of $\mathcal{H}''$. Our goal is to compute a vertex $u \in \mathcal{H}'$ and a vertex $v \in \mathcal{H}''$ such that the arc $A(u, v)$ is tangent to both $\mathcal{H}'$ and $\mathcal{H}''$. The algorithm is similar to that for computing a common tangent of two lower convex hulls that are separated by a vertical line; we briefly discuss it below.

Initially we set u to the rightmost vertex of $\mathcal{H}'$ and v the leftmost vertex of $\mathcal{H}''$. We keep moving u leftwards on $\mathcal{H}'$ until $A(u, v)$ is tangent to $\mathcal{H}'$ at u. Then we check whether

$A(u, v)$ is tangent to $\mathcal{H}''$ at v. If yes, then we are done. Otherwise, we keep moving v rightwards on $\mathcal{H}''$ until $A(u, v)$ is tangent to $\mathcal{H}''$ at v. Next we check whether $A(u, v)$ is tangent to $\mathcal{H}'$ at u. If yes, we are done. Otherwise, we move u leftwards again. We repeat this process and eventually a common tangent arc will be found. Clearly, the runtime is $O(|\mathcal{H}'| + |\mathcal{H}''|)$. □

With Lemma 5.5, the next lemma constructs the graph G.

Lemma 5.6. *The graph G can be constructed in $O(n \log n)$ time and $O(n)$ space.*

Proof. As the vertex set of G is Q, our goal is to construct all edges and store them in the adjacent-list structure, i.e., for each point $p \in Q$, construct the lists $L_l(p)$ and $L_r(p)$. To this end, our algorithm proceeds following the tree T in a bottom-up manner.

For each vertex $v \in T$, we define $G(v)$ as the graph G but only on the points of $Q(v)$. As such, $G(v)$ is G if v is the root and $G(v) = \emptyset$ if v is a leaf.

Consider an internal node v of T, with u and w as its left and right children, respectively. We assume that $G(u)$ and $G(w)$ have been computed, i.e., for each point p of $Q(u)$ (resp., $Q(v)$), we have two corresponding lists $L_l(p)$ and $L_r(p)$ with respect to $G(u)$ (resp., $G(v)$). Next, we construct $G(v)$ using $G(u)$ and $G(w)$.

Observe that $G(v)$ is the union of $G(u)$, $G(w)$, and the common tangent arc of $\mathcal{H}(u)$ and $\mathcal{H}(w)$, denoted by $A(p, q)$, with $p \in \mathcal{H}(u)$ and $q \in \mathcal{H}(w)$. Since $Q(u)$ and $Q(w)$ are separated by a vertical line, we can compute the arc $A(p, q)$ in $O(|Q(u)| + |Q(w)|)$ time by Lemma 5.5. Note that we can traverse on $\mathcal{H}(u)$ (resp., $\mathcal{H}(w)$) in constant time per vertex using the bottom edge pointers of vertices in $G(u)$ (resp., $G(w)$). Observe that the arc $A(p, q)$ must be the bottom edge in $L_r(p)$ as well as $L_l(q)$ in $G(v)$. As such, we simply add q to the bottom of the current list $L_r(p)$ and add p to the bottom of the current list $L_l(q)$, and also update the bottom edge pointers of p and q accordingly. In this way, $G(v)$ can be computed in $O(|Q(u)| + |Q(w)|)$ time, or in $O(|Q(v)|)$ time as $Q(v) = Q(u) \cup Q(w)$. Hence, the total time for constructing the graph G is $O(n \log n)$ and the space complexity is $O(n)$. □

5.3.3 Computing lower α-hull layers

We next use the graph G to compute the lower α-hull layers $\{\mathcal{H}_i\}$.

First of all, $\mathcal{H}_1$ can be obtained in $O(|\mathcal{H}_1|)$ time by using bottom edge pointers of G, say, starting from the leftmost point of Q, which is the leftmost vertex of $\mathcal{H}_1$, since arcs of $\mathcal{H}_1$ must be bottom edges of vertices of $\mathcal{H}_1$. Then, we remove vertices of $\mathcal{H}_1$ (along with their incident edges) from G. Using the new G, the second layer lower α-hull $\mathcal{H}_2$ can be computed in $O(|\mathcal{H}_2|)$ time similarly. We repeat this process until G becomes empty. The following lemma shows that removing a vertex from G can be done in $O(\log n)$ amortized time.

Lemma 5.7. *All point deletions in the entire algorithm can be done in $O(n \log n)$ time and $O(n)$ space.*

Proof. Suppose we want to delete a point p from G and p is a vertex of the lower α-hull of G. The deletion of p will result in the removal of all arcs of G connecting p. In addition, new arcs may be computed as well.

Let $\{v_1, v_2, ..., v_t\}$ be the list of the nodes of T encountered when traversing from the leaf node storing the point p to the root of T. The deletion of p may affect lower α-hulls $\mathcal{H}(v_i)$, for $i = 1, 2, ..., t$. We will update $G(v_i)$ (and thus $\mathcal{H}(v_i)$) for $i = 1, 2, ..., t$ in this order.

Consider a node v_i with $2 \leq i \leq t$. Note that v_{i-1} is a child of v_i. Let v refer to the child of v_i other than v_{i-1}. Depending on whether p is an endpoint of the arc $A(v_i)$ stored at v_i, i.e., the common tangent arc of $\mathcal{H}(v_{i-1})$ and $\mathcal{H}(v)$, there are two cases. If p is not an endpoint of $A(v_i)$, then removing p does not affect $A(v_i)$ as well as $A(v_j)$ for any $i + 1 \leq j \leq t$. Hence, in this case, we are done with deleting p. In the following, we focus on the case where p is an endpoint of $A(v_i)$ (see Fig. 5.13). Below we only discuss the case where p is the left endpoint of $A(v_i)$ since the other case is symmetric. Let c be the other endpoint of $A(v_i)$ and to be more informative we use $A(p, c)$ to refer to $A(v_i)$.

Note that each arc of $L_l(p) \cup L_r(p)$ is $A(v_j)$ for some $j \in [1, t]$. Let $A(a, p)$ be the last arc that has been processed due to the deletion of p with p as the right endpoint of the arc,

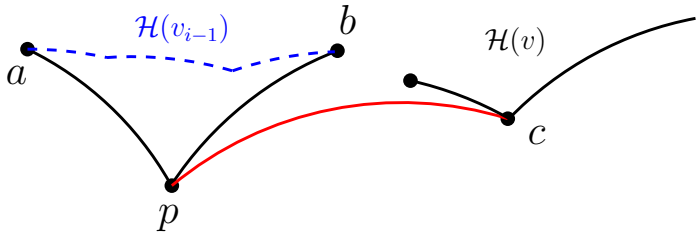


Fig. 5.13: p is an endpoint of $A(v_i)$, i.e., the common tangent arc (the red arc) of the new $\mathcal{H}(v_{i-1})$ and $\mathcal{H}(v)$.

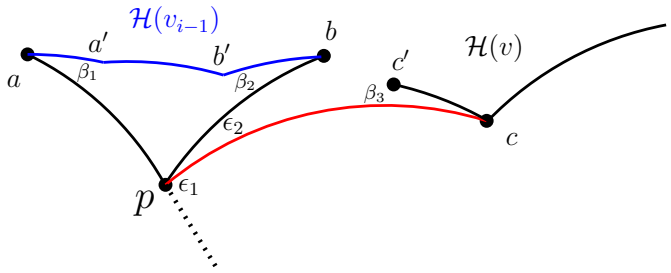


Fig. 5.14: Illustrating points a', b' and c', angles $\{\beta_1, \beta_2, \beta_3\}$ and $\{\epsilon_1, \epsilon_2\}$.

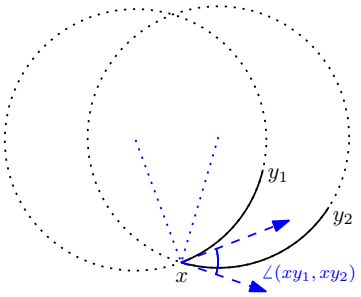


Fig. 5.15: Illustrating angle $\angle(xy_1, xy_2)$ of arcs $A(x, y_1)$ and $A(x, y_2)$. Blue rays with arrows are tangent rays of $A(x, y_1)$ and $A(x, y_2)$ at x.

and $A(p, b)$ the last arc that has been processed with p as the left endpoint of the arc (see Fig. 5.13). We assume that both a and b are well-defined (otherwise the algorithm is similar but simpler). Note that $A(a, p)$ and $A(p, b)$ are actually arcs of the old $\mathcal{H}(v_{i-1})$ before v_{i-1} is processed. Since we process nodes of T in a bottom-up matter, a and b can be accessed from $L_l(p)$ and $L_r(p)$ in constant time. Observe that the portion of the new lower α-hull $\mathcal{H}(v_{i-1})$ between a and b must lie above the "wedge" formed by $A(a, p)$ and $A(p, b)$ (see Fig. 5.13). Our goal is to compute a new common tangent arc $A(s, t)$ of the new $\mathcal{H}(v_{i-1})$ and $\mathcal{H}(v)$, with $s \in \mathcal{H}(v_{i-1})$ and $t \in \mathcal{H}(v)$, as follows.

Observe that s must lie between a and b on $\mathcal{H}(v_{i-1})$ and t is to the left of c on $\mathcal{H}(v)$. We define a' as the right adjacent vertex of a and b' as the left adjacent vertex of b on $\mathcal{H}(v_{i-1})$ (see Fig. 5.14). Let c' be the left adjacent vertex of c on $\mathcal{H}(v)$. The degenerate case in which $a' = b'$, or $a' = b$ and $b' = a$ can be handled trivially. Since p is a vertex of the current lower α-hull of G, arcs $A(a, p)$, $A(b, p)$, and $A(c, p)$ are bottom edges of $L_r(a)$,

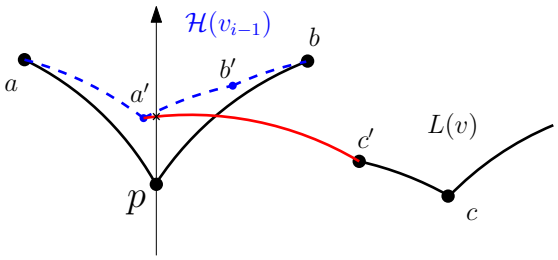


Fig. 5.16: Illustrating the case of pulling up p in which ϵ_1 becomes null.

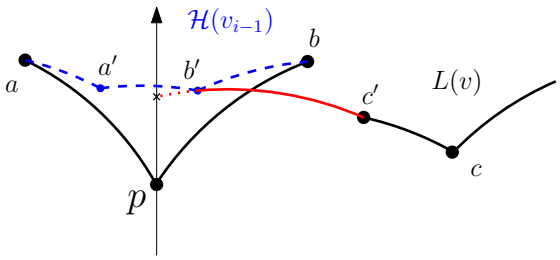


Fig. 5.17: Illustrating the case of pulling up p in which ϵ_2 becomes null.

$L_l(b)$, and $L_l(c)$, respectively. We can also access a', b', and c' in constant time.

To describe our algorithm for computing $A(s,t)$, we define *the angle* $\angle(xy_1, xy_2)$ of two arcs $A(x,y_1)$ and $A(x,y_2)$ as follows. For each $j = 1, 2$, define ρ_j as the ray from x toward y_j and tangent to the underlying disk of $A(x,y_j)$ at x. $\angle(xy_1, xy_2)$ is defined as the angle between ρ_1 and ρ_2 (see Fig. 5.15). Our algorithm considers the following five angles, $\beta_1 = \angle(aa', ap)$, $\beta_2 = \angle(bb', bp)$, $\beta_3 = \angle(cc', cp)$, $\epsilon_1 = \angle(pa'', pc)$, and $\epsilon_2 = \angle(pb, pc)$, where a'' is a point on the extension of arc $A(a,p)$ (see Fig. 5.14).

Our algorithm for computing $A(s,t)$ can be viewed as a process of "pulling up" p vertically until p disappears in the new lower α-hull $\mathcal{H}(v_i)$. This happens when one of $\{\epsilon_1, \epsilon_2\}$ becomes null (see Fig. 5.16 and 5.17). If one of the angles of $\{\beta_1, \beta_2, \beta_3\}$ becomes null, then we will update x and x', $x \in \{a, b, c\}$ accordingly to obtain new β-angles. More specifically, if $\angle(xx', xp)$, $x \in \{a, b, c\}$ becomes null, then we reset x to x', and reset x' to the left (if $x \in \{b, c\}$) or right (if $x \in \{a\}$) neighbor of the old x'. For the purpose of time analysis, we say that the old x is *wrapped*. We can avoid calculating those five angles by computing the intersections a^*, b^*, and c^* of the vertical line through p with extensions of arcs $A(a,a')$, $A(b,b')$ and $A(c,c')$, respectively (see Fig. 5.18). The lowest point of a^*, b^*, and c^* is the next candidate location of p. Before moving p to the next location, we check whether $\{a, p, c\}$ or $\{p, b, c\}$ will be on the same unit circle during the movement of p, and a positive answer implies that either ϵ_1 or ϵ_2 is null. We iterate this process until one of ϵ_1 and ϵ_2 is null. Once the common tangent arc $A(s,t)$ is computed, we proceed on processing v_{i+1}. Note that p is in $L_l(c)$ and is actually the bottom edge since p is a vertex of the lower

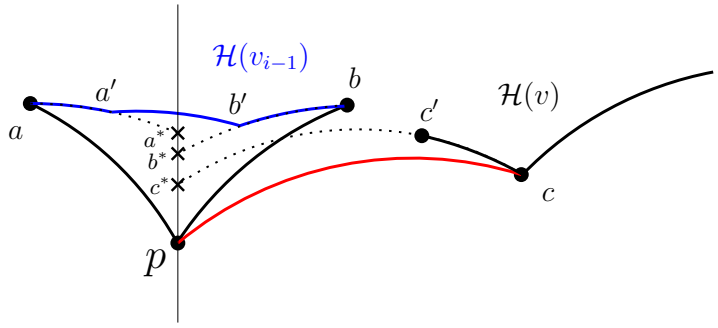


Fig. 5.18: Illustrating the definitions of a^*, b^*, and c^*.

α-hull of G; as such, we can remove p from $L_l(c)$ and reset its bottom edge in constant time.

The running time of the algorithm is linear in the number of wrapped vertices on $\mathcal{H}(v_{i-1})$ and $\mathcal{H}(v)$. If a vertex u is wrapped by a or c, then u becomes a vertex on the new $\mathcal{H}(v_i)$. We call this wrapping step a *promotion* (because u used to be a vertex of $\mathcal{H}(v_{i-1})$ and not a vertex of $\mathcal{H}(v_i)$, but now is "promoted" to be a vertex of $\mathcal{H}(v_i)$). Since the height of T is $O(\log n)$, the total number of promotions for deleting all points $p \in G$ is bounded by $O(n \log n)$. On the other hand, if a vertex u is wrapped by b, we call it a *confirmation*. A critical observation is that the previous wrapping on u during the deletion of p must be a promotion (i.e., during processing v_{i-1}, u was wrapped as a promotion). Consequently, any confirmation must be immediately preceded by a promotion. As such, the total number of confirmations for deleting all points $p \in G$ is no more than that of promotions, which is $O(n \log n)$. Therefore, the overall time of the algorithm for deleting all points $p \in G$ is bounded by $O(n \log n)$. The space complexity of the algorithm is $O(n)$. □

With Lemma 5.7, Theorem 5.2 is proved.

5.4 Concluding remarks

In this chapter, we proposed a simple algorithm for the unit-disk range reporting problem and the performance of our algorithm matches that of the previously best result. Our techniques may be extended to solve other related problems. We demonstrate two exemplary problems below.

Outside-unit-disk range reporting queries. This is a symmetric problem to the unit-disk range reporting problem and the problem aims to report all points of P that are outside of a query unit disk. To solve this outside version of the problem, we modify our approach as follows. We also build a grid as in Section 5.2. Given a query unit disk D_q whose center is q, we first identify the cell C of the grid which contains q. Then points of P that lie in cells of $\mathcal{C} \setminus N(C)$ can be reported directly since these points cannot be in D_q. Next, we solve the line-separable version of this problem within a cell of $N(C)$ and follow the notation in previous sections. The target is to report all arcs of $\mathcal{A}$ that are above q. The basic idea is to consider all layers of upper envelopes of arcs of $\mathcal{A}$. We construct these layers of upper envelopes by computing upper α-hull layers of points of Q, where $\alpha = 1$. This is symmetric to the algorithm of computing lower α-hull layers in Section 5.3, where $\alpha = -1$. Then we build a fractional cascading data structure on the layers of upper envelopes similarly to our method for layers of lower envelopes. To answer the query for D_q, we perform a binary search on layers of upper envelopes and find out the arc that spans point q in each upper envelope. We start from the highest upper envelope and move downwards one by one until q is above a layer of upper envelopes. For each layer that is above point q, we traverse from the arc that spans q and move leftwards and rightwards, respectively, to check whether each arc is also above q. Once an arc is encountered that is below q, we stop traversing along this direction. Points of P corresponding to the arcs that are above q are reported. The total query time is $O(\log n + k)$, where k is the output size. As before, the preprocessing takes $O(n \log n)$ time and $O(n)$ space.

Unit-disk range emptiness queries. The problem is to determine whether there is at least one point of P lying in a query unit disk. We follow our method for the original unit-disk range reporting queries but only maintain the lower envelope of $\mathcal{A}$ (i.e., no need to compute all layers of lower envelopes and thus the algorithm becomes much simpler). A point of P is in the query unit disk if and only if the disk center q is above the lower envelope, which can be determined by performing binary search with q on the lower envelope of $\mathcal{A}$. The query time is thus $O(\log n)$. The preprocessing still takes $O(n \log n)$ time and $O(n)$

space. In addition, following the similar approach (e.g., maintaining the upper envelope of $\mathcal{A}$ instead), we can also solve *outside-unit-disk range emptiness queries*, i.e., deciding whether a point of P is outside a query unit disk. The complexities are the same as above.

CHAPTER 6
IMPROVED ALGORITHMS FOR DISTANCE SELECTION AND RELATED PROBLEMS

6.1 Introduction

We improve the distance selection algorithm and give two algorithmic frameworks that can solve geometric optimization problems involving interpoint distances in a point set in the plane, e.g., two-sided and one-sided discrete Fréchet distance with shortcuts. The result in this chapter has been submitted to a conference and is now still under review.

6.1.1 Problem definitions and our results

We propose new techniques for solving geometric optimization problems involving interpoint distances in a point set in the plane. More specifically, the optimal objective value of these problems is equal to the (Euclidean) distance of two points in the set. Our techniques usually yield improvements over the previous work by at least a logarithmic factor (and sometimes a polynomial factor).

The first problem we consider is the *distance selection* problem: Given a set P of n points in the plane and an integer $1 \leq k \leq \binom{n}{2}$, the problem asks for the k-th smallest interpoint distance among all pairs of points of P. The problem can be easily solved in $O(n^2)$ time. The first subquadratic time algorithm was given by Chazelle [93]; the algorithm runs in $O(n^{9/5}\log^{4/5} n)$ time and is based on Yao's technique [94]. Later, Agarwal, Aronov, Sharir, and Suri [95] gave a better algorithm of $O(n^{3/2}\log^{5/2} n)$ time and subsequently Goodrich [96] solved the problem in $O(n^{4/3}\log^{8/3} n)$ time. Katz and Sharir [30] finally presented an $O(n^{4/3}\log^2 n)$ time algorithm. All above are deterministic algorithms. Several randomized algorithms have also been proposed for the problem. The randomized algorithm of [95] runs in $O(n^{4/3}\log^{8/3} n)$ expected time. Matousek [97] gave another

randomized algorithm of $O(n^{4/3}\log^{2/3} n)$ expected time. Very recently, Chan and Zheng proposed a randomized algorithm of $O(n^{4/3})$ expected time (see the arXiv version of [31]). Also, the time complexity can be made as a function of k. In particular, Chan's randomized techniques [32] solved the problem in $O(n\log n + n^{2/3}k^{1/3}\log^{5/3} n)$ expected time and Wang [33] recently improved the algorithm to $O(n\log n + n^{2/3}k^{1/3}\log n)$ expected time; these algorithms are particularly interesting when k is relatively small.

In this chapter, we present a new deterministic algorithm that solves the distance selection problem in $O(n^{4/3}\log n)$ time. Albeit slower than the randomized algorithm of Chan and Zheng [31], our algorithm is the first progress on the deterministic solution since the work of Katz and Sharir [30] published 25 years ago (30 years if we consider their conference version in SoCG 1993).

One technique we introduce is an algorithm for solving the following *partial batched range searching problem*.

Problem 6.1. (Partial batched range searching) *Given a set A of m points and a set B of n points in the plane and an interval $(\alpha, \beta]$, one needs to construct two collections of edge-disjoint complete bipartite graphs $\Gamma(A, B, \alpha, \beta) = \{A_t \times B_t \mid A_t \subseteq A, B_t \subseteq B\}$ and $\Pi(A, B, \alpha, \beta) = \{A'_s \times B'_s \mid A'_s \subseteq A, B'_s \subseteq B\}$ such that the following two conditions are satisfied.*

1. *For each pair $(a, b) \in A_t \times B_t \in \Gamma(A, B, \alpha, \beta)$, the (Euclidean) distance $\|ab\|$ between points $a \in A_t$ and $b \in B_t$ is in $(\alpha, \beta]$.*

2. *For any two points $a \in A$ and $b \in B$ with $\|ab\| \in (\alpha, \beta]$, either $\Gamma(A, B, \alpha, \beta)$ has a unique graph $A_t \times B_t$ that contains (a, b) or $\Pi(A, B, \alpha, \beta)$ has a unique graph $A'_s \times B'_s$ that contains (a, b).*

In other words, the two collections Γ and Π together record all pairs (a, b) of points $a \in A$ and $b \in B$ whose distances are in $(\alpha, \beta]$. While all pairs of points recorded in Γ have their distances in $(\alpha, \beta]$, this may not be true for Π. For this reason, we sometimes call the point pairs recorded in Π uncertain pairs.

Note that if context is clear, we sometimes use Γ and Π to refer to $\Gamma(A, B, \alpha, \beta)$ and $\Pi(A, B, \alpha, \beta)$, respectively. Also, for short, we use BRS to refer to batched range searching.

In the traditional BRS, which has been studied with many applications, e.g., [18,36,40], the collection Π is $\emptyset$ (and thus Γ itself satisfies the two conditions in Problem 6.1); for differentiation, we refer to this case as the *complete BRS*. The advantage of the partial problem over the complete problem is that the partial problem can usually be solved faster, with a sacrifice that some uncertain pairs (i.e., those recorded in Π) are left unresolved. As will be seen later, in typical applications the number of those uncertain pairs can be made small enough so that they can be handled easily without affecting the overall runtime of the algorithm. More specifically, we derive an algorithm to compute a solution for the partial BRS, whose runtime is controlled by a parameter (roughly speaking, the runtime increases as the graph sizes of Π decreases). Previously, Katz and Shair [30] gave an algorithm for the complete problem. Our solution, albeit for the more general partial problem, even improves their algorithm by roughly a logarithmic factor when applied to the complete case.

On the one hand, our partial BRS solution helps achieve our new result for the distance selection problem. On the other hand, combining some techniques for the latter problem, we propose a general algorithmic framework that can be used to solve any geometric optimization problems that involve interpoint distances of a set of points in the plane. Consider such a problem whose optimal objective value (denoted by δ^*) is equal to the distance of two points of a set P of n points in the plane. Assume that the decision problem (i.e., given δ, decide whether $\delta \geq \delta^*$) can be solved in T_D time. A straightforward algorithm for computing δ^* is to use the distance selection algorithm and the decision algorithm to perform binary search on interpoint distances of all pairs of points of P; the algorithm runs in $O(\log n)$ iterations and each iteration takes $O(n^{4/3} \log n + T_D)$ time (if we use our new distance selection algorithm). As such, the total runtime is $O((n^{4/3} \log n + T_D) \log n)$. Using our new framework, the runtime can be bounded by $O((n^{4/3} + T_D) \log n)$, which is faster when $T_D = o(n^{4/3} \log n)$.

One application of this new framework is the *two-sided discrete Fréchet distance with*

shortcuts problem, or *two-sided DFD* for short. Fréchet distance is used to measure the similarity between two curves and many of its variations have been studied, e.g., [34–39]. To reduce the impact of outliers between two (sampled) curves, discrete Fréchet distance with shortcuts was proposed [36,39]. If outliers of only one curve need to be taken care of, it is called *one-sided DFD*; otherwise it is *two-sided DFD*. Avraham, Filtser, Kaplan, Katz, and Sharir [36] solved the two-sided DFD in $O((m^{2/3}n^{2/3}+m+n)\log^3(m+n))$, where m and n are the numbers of vertices of the two input curves, respectively. Using our new framework, we improve their algorithm to $O((m^{2/3}n^{2/3}\cdot 2^{O(\log^*(m+n))}+m\log n+n\log m)\log(m+n))$ time, an improvement of roughly $O(\log^2(m+n))$.

For the one-sided DFD, the authors [36] gave a randomized algorithm of $O((m+n)^{6/5+\epsilon})$ expected time, for any constant $\epsilon>0$. Using our solution to the partial BRS, we improve their algorithm to $O((m+n)^{6/5}\log^{8/5}(m+n))$ expected time. Based on the techniques of [36], Katz and Sharir [40] proposed an algorithmic framework for solving geometric optimization problems that involve interpoint distances in a point set. Consider such a problem whose optimal objective value (denoted by δ^*) is equal to the distance of two points of a set P of n points in the plane. The framework has two main procedures. The first procedure is to compute an interval that contains δ^* and with high probability at most L interpoint distances of P. Using the interval and a *bifurcation tree* technique, the second main procedure finally computes δ^*. Assuming that the decision problem can be solved in T_D time, the first main procedure takes $O(n^{4/3+\epsilon}/L^{1/3}+T_D\cdot\log n\cdot\log\log n)$ expected time and the second one runs in $O(L^{1/2}\cdot T_D\cdot\log n)$ time, resulting an algorithm of $O(n^{4/3+\epsilon}/L^{1/3}+T_D\cdot\log n\cdot\log\log n+L^{1/2}\cdot T_D\cdot\log n)$ expected time in total [36,40]. Using our partial BRS solution, we improve the first main procedure to $O(n^{4/3}/L^{1/3}\cdot\log^2 n+T_D\cdot\log n\cdot\log\log n)$ expected time, which eliminates the $O(n^\epsilon)$ factor. Thus, the total expected time of the framework becomes $O(n^{4/3}/L^{1/3}\cdot\log^2 n+T_D\cdot\log n\cdot\log\log n+L^{1/2}\cdot T_D\cdot\log n)$. Our result for the one-sided DFD is a direct application of this framework. More specifically, since $T_D=O(m+n)$ [36], we set $L=(m+n)^{2/5}\log^{6/5}(m+n)$ and replace n by $(m+n)$ in the above time complexity as there are two parameters m and n in the problem.

We demonstrate two more applications of the framework where our new techniques lead to improved results over the previous work: the *reverse shortest paths in unit-disk graphs* and its weighted case. Given a set P of n points in the plane and a parameter $\delta > 0$, the unit-disk graph $G_\delta(P)$ is an undirected graph whose vertex set is P such that an edge connects two points $p, q \in P$ if the (Euclidean) distance between p and q is at most δ. In the unweighted (resp., weighted) case, the weight of each edge is equal to 1 (resp., the distance between the two vertices). Given set P, two points $s, t \in P$, and a parameter λ, the problem is to compute the smallest δ^* such that the shortest path length between s and t in $G_{\delta^*}(P)$ is at most λ.

Deterministic algorithms of $O(n^{4/3}\log^{7/4} n)$ and $O(n^{4/3}\log^{5/2} n)$ times are known for the unweighted and weighted problems, respectively [17, 18]. The decision problem for the unweighted case can be solved in $O(n)$ time (after points of P are sorted) [13] while the weighted case can be solved in $O(n\log^2 n)$ time [1]. As such, using their framework, Katz and Sharir [40] solved both problems in $O(n^{6/5+\epsilon})$ expected time (by setting $L = n^{2/5}$). With our improvement to the framework, we can now solve the unweighted problem in $O(n^{6/5}\log^{8/5} n)$ expected time (by setting $L = n^{2/5}\log^{6/5} n$) and solve the weighted case in $O(n^{6/5}\log^{12/5} n)$ expected time (by setting $L = n^{2/5}/\log^{6/5} n$).

In summary, we propose two algorithmic frameworks for solving geometric optimization problems that involve interpoint distances in a set of points in the plane. The first framework is deterministic while the second one is randomized. The first framework is mainly useful when the decision algorithm time T_D is relatively large (e.g., close to $O(n^{4/3})$) while the second one is more interesting when T_D is relatively small (e.g., near linear). Both frameworks rely on our solution to the partial BRS problem. As optimization problems involving interpoint distances are very common in computational geometry, we believe our techniques will find more applications in future.

Outline. The rest of the chapter is organized as follows. Section 6.2 presents our algorithm for the partial BRS. The algorithm for the distance selection problem is described in Section 6.3. The two-sided DFD problem is solved in Section 6.4, where we also propose

our first algorithmic framework. The one-sided DFD problem and our second algorithmic framework are discussed in Section 6.5.

6.2 Partial batched range searching

In this section, we present our solution to the partial BRS problem, i.e., Problem 6.1. We follow the notation in the statement of Problem 6.1. In particular, $m = |A|$ and $n = |B|$.

For any set P of points and a compact region R in the plane, let $P(R)$ denote the subset of points of P in R, i.e., $P(R) = P \cap R$. For any point p in the plane, with respect to the interval $(\alpha, \beta]$ in Problem 6.1, let D_p denote the annulus centered at p and having radii α and β (e.g., see Fig. 6.1); so D_p has an inner boundary circle of radius α and an outer boundary circle of radius β. We assume that D_p includes its outer boundary circle but not its inner boundary circle. In this way, a point q is in D_p if and only if $\|pq\| \in (\alpha, \beta]$. Define $\mathcal{D}$ as the set of all annuli D_p for all points $p \in A$. Define $\mathcal{C}$ to be the set of boundary circles of all annuli of $\mathcal{D}$. Hence, $\mathcal{C}$ consists of $2m$ circles. For any compact region R in the plane, let $\mathcal{C}_R$ denote the subset of circles of $\mathcal{C}$ that intersect the relative interior of R.

An important tool we use is the cuttings [98]. For a parameter $1 \leq r \leq n$, a $(1/r)$-cutting Ξ of size $O(r^2)$ for $\mathcal{C}$ is a collection of $O(r^2)$ constant-complexity cells whose union covers the plane such that the interior of each cell $\sigma \in \Xi$ is intersected by at most m/r circles in $\mathcal{C}$, i.e., $|\mathcal{C}_\sigma| \leq m/r$.

We actually use *hierarchical cuttings* [98]. We say that a cutting Ξ' *c-refines* a cutting Ξ if each cell of Ξ' is contained in a single cell of Ξ and every cell of Ξ contains at most c cells of Ξ'. Let Ξ_0 denote the cutting whose single cell is the whole plane. Then we define cuttings $\{\Xi_0, \Xi_1, ..., \Xi_k\}$, in which each Ξ_i, $1 \leq i \leq k$, is a $(1/\rho^i)$-cutting of size $O(\rho^{2i})$ that c-refines Ξ_{i-1}, for two constants ρ and c. By setting $k = \lceil \log_\rho r \rceil$, the last cutting Ξ_k is a $(1/r)$-cutting. The sequence $\{\Xi_0, \Xi_1, ..., \Xi_k\}$ of cuttings is called a hierarchical $(1/r)$-cutting of $\mathcal{C}$. For a cell σ' of Ξ_{i-1}, $1 \leq i \leq k$, that fully contains cell σ of Ξ_i, we say that σ' is the *parent* of σ and σ is a *child* of σ'. Thus the hierarchical $(1/r)$-cutting can be viewed as a tree structure with Ξ_0 as the root.

A hierarchical $(1/r)$-cutting of $\mathcal{C}$ can be computed in $O(mr)$ time, e.g., by the algorithm

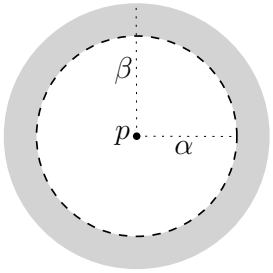


Fig. 6.1: Illustrating an annulus D_p (the grey region).

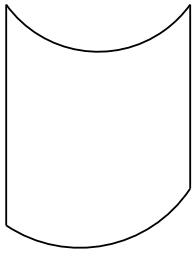


Fig. 6.2: Illustrating a pseudo-trapezoid.

in [33], which adapts Chazelle's algorithm [98] for hyperplanes. The algorithm also produces the subset $\mathcal{C}_\sigma$ for all cells $\sigma \in \Xi_i$ for all $i = 0, 1, \ldots, k$, implying that the total size of these subsets is bounded by $O(mr)$. In particular, each cell of the cutting produced by the algorithm of [33] is a *pseudo-trapezoid* that is bounded by two vertical line segments from left and right, an arc of a circle of $\mathcal{C}$ from top, and an arc of a circle of $\mathcal{C}$ from bottom (e.g., see Fig. 6.2).

Using the cutting, we obtain the following solution to the partial BRS problem.

Lemma 6.1. *For any r with $1 \leq r \leq \min\{m^{1/3}, n^{1/3}\}$, we can compute in $O(mr \log r + nr)$ time two collections $\Gamma(A, B, \alpha, \beta) = \{A_t \times B_t \mid A_t \subseteq A, B_t \subseteq B\}$ and $\Pi(A, B, \alpha, \beta) = \{A'_s \times B'_s \mid A'_s \subseteq A, B'_s \subseteq B\}$ of edge-disjoint complete bipartite graphs that satisfy the conditions of Problem 6.1, with the following complexities: (1) $|\Gamma| = O(r^4)$; (2) $\sum_t |A_t|, \sum_t |B_t| = O(mr \log r + nr)$; (3) $|\Pi| = O(r^4)$; (4) $|A'_s| = O(m/r^3)$ and $|B'_s| = O(n/r^3)$ for each $A'_s \times B'_s \in \Pi$; (5) the number of pairs of points recorded in Π is $O(r^4 \cdot m/r^3 \cdot n/r^3) = O(mn/r^2)$.*

Proof. We begin with constructing a hierarchical $(1/r)$-cutting $\{\Xi_0, \Xi_1, ..., \Xi_k\}$ for $\mathcal{C}$, which takes $O(mr)$ time as discussed above. We use Ξ to refer to the set of all cells σ in all cuttings Ξ_i, $0 \leq i \leq k$. Next we compute the set $B(\sigma)$ for each cell σ in the cutting (recall that $B(\sigma)$ refers to the subset of points of B inside σ; we call $B(\sigma)$ a *canonical subset*). This can be done in $O(n \log r)$ time in a top-down manner by processing each point of B individually. Specifically, for each point $p \in B$, suppose we know that p is in σ' for a cell σ' in Ξ_{i-1} (which is true initially when $i = 1$ as Ξ_0 has a single cell that is the entire plane).

By examining each child of σ' we can find in $O(1)$ time the cell σ of Ξ_i that contains p and then we add p to $B(\sigma)$. Since $k = \Theta(\log r)$, each point of B is stored in $O(\log r)$ canonical subsets and the total size of all canonical subsets $B(\sigma)$ for all cells $\sigma \in \Xi$ is $O(n \log r)$.

Next, for each cell σ of Ξ, we compute another canonical subset $A_\sigma \subseteq A$. Specifically, a point $p \in A$ is in A_σ if the annulus D_p contains σ but not σ's parent. The subsets A_σ for all cells σ of Ξ can be computed in $O(mr)$ time. Indeed, recall that the cutting algorithm already computes $\mathcal{C}_\sigma$ for all cells $\sigma \in \Xi$. For each Ξ_{i-1}, $1 \leq i \leq k$, for each cell σ' of Ξ_{i-1}, we consider each circle $C \in \mathcal{C}_{\sigma'}$. Let p be the point of A such that C is a bounding circle of the annulus D_p. For each child σ of σ', if D_p fully contains σ, then we add p to A_σ. In this way, A_σ for all cells σ of Ξ can be computed in $O(mr)$ time since $\sum_{0 \leq i \leq k} \sum_{\sigma' \in \Xi_i} |\mathcal{C}_{\sigma'}| = O(mr)$ and each cell σ' has $O(1)$ children. As such, the total size of A_σ for all cells $\sigma \in \Xi$ is $O(mr)$.

By definition, for each cell $\sigma \in \Xi$, for any point $a \in A_\sigma$ and any point $b \in B(\sigma)$, we have $\|ab\| \in (\alpha, \beta]$. As such, we return $\{A_\sigma \times B(\sigma) \mid \sigma \in \Xi\}$ as a subcollection of $\Gamma(A, B, \alpha, \beta)$ to be computed for the lemma. Note that the complete bipartite graphs of $\{A_\sigma \times B(\sigma) \mid \sigma \in \Xi\}$ are edge-disjoint. The size of the subcollection is equal to the number of cells of the hierarchical cutting, which is $O(r^2)$. Also, we have shown above that $\sum_{\sigma \in \Xi} |A_\sigma| = O(mr)$ and $\sum_{\sigma \in \Xi} |B(\sigma)| = O(n \log r)$.

For each cell σ of the last cutting Ξ_k, we have $|\mathcal{C}_\sigma| \leq m/r$. Let $\hat{A}_\sigma$ denote the subset of points $p \in A$ such that D_p has a bounding circle in $\mathcal{C}_\sigma$. We do not know whether distances between points of $\hat{A}_\sigma$ and points of $B(\sigma)$ are in $(\alpha, \beta]$ or not. If $|B(\sigma)| > n/r^2$, then we arbitrarily partition $B(\sigma)$ into subsets of size between $n/(2r^2)$ and n/r^2. We call these subsets *standard subsets* of $B(\sigma)$. Since $|B| = n$ and we have $O(r^2)$ cells in cutting Ξ_k, the number of standard subsets of all cells of Ξ_k is $O(r^2)$. For each standard subset $\hat{B}(\sigma) \subseteq B(\sigma)$, we form a pair $(\hat{A}_\sigma, \hat{B}(\sigma))$ as an "unsolved" *subproblem*. Then we have $O(r^2)$ subproblems. Note that $|\hat{A}_\sigma| \leq m/r$ and $|\hat{B}(\sigma)| \leq n/r^2$. If we apply the same algorithm recursively on each subproblem, then we have the following recurrence relation (which holds

for any $1 \leq r \leq m$):

$$T(m,n) = O(mr + n\log r) + O(r^2) \cdot T(\frac{m}{r}, \frac{n}{r^2}) \tag{6.1}$$

Note that if we use $T(m,n)$ to represent the total size of A_t and B_t of all complete bipartite graphs $A_t \times B_t$ in the subcollection of $\Gamma(A,B,\alpha,\beta)$ that have been produced as above, then we have the same recurrence as above. If $N(m,n)$ denotes the number of these graphs, then we have the following recurrence:

$$N(m,n) = O(r^2) + O(r^2) \cdot N(\frac{m}{r}, \frac{n}{r^2})$$

We now solve the problem in a "dual" setting by switching the roles of A and B, i.e., define annuli centered at points of B and compute the hierarchical cutting for their bounding circles. Then, symmetrically we have the following recurrences (which holds for any $1 \leq r \leq n$):

$$T(m,n) = O(nr + m\log r) + O(r^2) \cdot T(\frac{m}{r^2}, \frac{n}{r}) \tag{6.2}$$

$$N(m,n) = O(r^2) + O(r^2) \cdot N(\frac{m}{r^2}, \frac{n}{r})$$

By applying (6.2) to each subproblem of (6.1) using the same parameter r and we can obtain the following recurrence:

$$T(m,n) = O(mr\log r + nr) + O(r^4) \cdot T(\frac{m}{r^3}, \frac{n}{r^3})$$

Similarly, we have

$$N(m,n) = O(r^4) + O(r^4) \cdot N(\frac{m}{r^3}, \frac{n}{r^3})$$

The above recurrences tell us that in $O(mr\log r + nr)$ time we can compute a collection

of $O(r^4)$ edge-disjoint complete bipartite graphs $A_t \times B_t$ with $A_t \subseteq A$ and $B_t \subseteq B$ such that for any two points $a \in A_t$ and $b \in B_t$ their distance $\|ab\|$ lies in $(\alpha, \beta]$. Further, the size of all such A_t's and B_t's is bounded by $O(mr \log r + nr)$. We return the above collection as $\Gamma(A, B, \alpha, \beta)$ for the lemma.

In addition, we have also $O(r^4)$ graphs $A'_s \times B'_s$ with $A'_s \subseteq A$ and $B'_s \subseteq B$ corresponding to the unsolved subproblems $T(m/r^3, n/r^3)$ and we do not know whether $\|ab\| \in (\alpha, \beta]$ for points $a \in A'_s$ and $b \in B'_s$. We return the collection of all such graphs as $\Pi(A, B, \alpha, \beta)$ for the lemma. Hence, $|\Pi(A, B, \alpha, \beta)| = O(r^4)$, and $|A'_s| \leq m/r^3$ and $|B'_s| \leq n/r^3$ for each graph $A'_s \times B'_s$ in the collection. The number of pairs of points recorded in $\Pi(A, B, \alpha, \beta)$ is $O(|\Pi(A, B, \alpha, \beta)| \cdot m/r^3 \cdot n/r^3)$, which is $O(mn/r^2)$. This proves the lemma. □

The following theorem solves the complete BRS problem by running the algorithm of Lemma 6.1 recursively until the problem size becomes $O(1)$.

Theorem 6.1. *We can compute in $O(m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m \log n + n \log m)$ time a collection $\Gamma(A, B, \alpha, \beta) = \{A_t \times B_t \mid A_t \subseteq A, B_t \subseteq B\}$ of edge-disjoint complete bipartite graphs that satisfy the conditions of Problem 6.1 (with $\Pi(A, B, \alpha, \beta) = \emptyset$), with the following complexities: (1) $|\Gamma| = O(m^{2/3}n^{2/3} \cdot \log^*(m+n) + m + n)$; (2) $\sum_t |A_t|, \sum_t |B_t| = O(m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m \log n + n \log m)$.*

Proof. To solve the complete BRS problem, the main idea is to apply the recurrence (6.2) recursively until the size of each subproblem becomes $O(1)$. We first consider the symmetric case where $m = n$. By setting $r = n^{1/3}/\log n$ and applying (6.2) with $m = n$, we obtain the following

$$T(n, n) = O(n^{4/3}) + O(n^{4/3}/\log^4 n) \cdot T(\log^3 n, \log^3 n) \tag{6.3}$$

Similarly, we have

$$N(n, n) = O(n^{4/3}/\log^4 n) + O(n^{4/3}/\log^4 n) \cdot N(\log^3 n, \log^3 n) \tag{6.4}$$

The recurrences solve to $T(n,n) = n^{4/3} \cdot 2^{O(\log^* n)}$ and $N(n,n) = O(n^{4/3} \cdot \log^* n)$. This means that in $n^{4/3} \cdot 2^{O(\log^* n)}$ time we can compute a collection $\Gamma(A,B,\alpha,\beta) = \{A_t \times B_t \mid A_t \subseteq A, B_t \subseteq B\}$ of $O(n^{4/3} \log^* n)$ edge-disjoint complete bipartite graphs, with $\sum_t |A_t|, \sum_t |B_t| = n^{4/3} \cdot 2^{O(\log^* n)}$, and it satisfies the conditions of Problem 6.1 with $\Pi(A,B,\alpha,\beta) = \emptyset$.

We now consider the asymmetric case, i.e., $m \neq n$. We first assume $m \leq n$. Depending on whether $n < m^2$, there are two cases.

1. If $n < m^2$, we set $r = n/m$ so that $m/r = n/r^2$. We apply recurrence (6.1) and solve each subproblem of size $(m/r, n/r^2) = (m^2/n, m^2/n)$ by our above algorithm for the symmetric case, which results in $T(m,n) = O(n \log m + m^{2/3}n^{2/3} \cdot 2^{O(\log^* n)})$. Similarly, the number of graphs in the produced collection is $O(m^{2/3}n^{2/3} \log^* n)$ and the total size of vertex sets of these graphs is $O(n \log m + m^{2/3}n^{2/3} \cdot 2^{O(\log^* n)})$.

2. If $n \geq m^2$, then we simply apply recurrence (6.1) with $r = m$ and obtain $T(m,n) = O(m^2 + n \log m) + O(m^2) \cdot T(1, n/m^2)$. Note that $T(1, n/m^2)$ can be solved in $O(n/m^2)$ time by brute force. Therefore, the recurrence solves to $T(m,n) = O(m^2 + n \log m)$, which is $O(n \log m)$ as $n \geq m^2$. Similarly, the number of of complete bipartite graphs in the generated collection is $O(n)$, and the total size of vertex sets of these graphs is $O(n \log m)$.

In summary, if $m \leq n$, we can solve the complete BRS problem in $O(n \log m + m^{2/3}n^{2/3} \cdot 2^{O(\log^* n)})$ time, by generating $O(m^{2/3}n^{2/3} \log^* n + n)$ complete bipartite graphs whose vertex set size is bounded by $O(n \log m + m^{2/3}n^{2/3} \cdot 2^{O(\log^* n)})$.

If $m > n$, then the analysis is symmetric with the notation m and n flipped in the above complexities. The theorem is thus proved. □

For comparison, Katz and Shair [30] solved the complete BRS problem in $O((m^{2/3}n^{2/3} + m + n) \log m)$ time by producing $O(m^{2/3}n^{2/3} + m + n)$ complete bipartite graphs whose total vertex set size is $O((m^{2/3}n^{2/3} + m + n) \log m))$. Our result improves their runtime and vertex set size by almost a logarithmic factor with slight more graphs produced. One may

wonder whether Chan and Zheng's recent techniques [31] could be used to reduce the factor $2^{O(\log^* n)}$. It is not clear to us whether this is possible. Indeed, Chan and Zheng's techniques are mainly for solving point locations in line arrangements and in their problem they only need to locate a single cell of the arrangement that contains a point. In the point location step of our problem (i.e., computing the canonical sets $B(\sigma)$ in Lemma 6.1), however, we have to use hierarchical cutting and construct the canonical sets $B(\sigma)$ for each cell σ that contains the point in every cutting Ξ_i, $1 \leq i \leq k$ (i.e., our problem needs to locate $O(\log r)$ cells per point).

6.3 Distance selection

In this section, we present our algorithm for the distance selection problem. Let P be a set of n points in the plane. Define $\mathcal{E}(P)$ as the set of distances of all pairs of points of P. Given an integer $1 \leq k \leq \binom{n}{2}$, the problem is to find the k-th smallest value in $\mathcal{E}(P)$, denoted by δ^*.

Given any δ, the *decision problem* is to determine whether $\delta \geq \delta^*$. Wang [33] recently gave an $O(n^{4/3})$ time algorithm that can compute the number of values of $\mathcal{E}(P)$ at most δ, denoted by k_δ. Observe that $\delta \geq \delta^*$ if and only if $k_\delta \geq k$. Thus, using Wang's algorithm [33], the decision problem can be solved in $O(n^{4/3})$ time. We should point out that the $O(n^{4/3}\log^2 n)$ time algorithm of Katz and Shair [30] for computing δ^* utilizes a decision algorithm of $O(n^{4/3}\log n)$ time. However, even if we replace their decision algorithm by Wang's $O(n^{4/3})$ time algorithm, the runtime of the overall algorithm for computing δ^* is still $O(n^{4/3}\log^2 n)$ because other parts of the algorithm dominate the total time. To reduce the overall time to $O(n^{4/3}\log n)$, new techniques are needed, in addition to using the faster $O(n^{4/3})$ time decision algorithm. These new techniques include, for instance, Lemma 6.1 for the partial BRS problem, as will be seen below.

Before presenting the details of our algorithm, we first prove the following lemma, which is critical to our algorithm and is obtained by using Lemma 6.1.

Lemma 6.2. *Given an interval $(\alpha, \beta]$, Problem 6.1 with $A = P$ and $B = P$ can be*

solved in $O(n^{4/3})$ *time by computing two collections* $\Gamma(P, P, \alpha, \beta) = \{A_t \times B_t \mid A_t, B_t \subseteq P\}$ *and* $\Pi(P, P, \alpha, \beta) = \{A'_s \times B'_s \mid A'_s, B'_s \subseteq P\}$ *with the following complexities: (1)* $|\Gamma| = O(n^{4/3}/\log^4 \log n)$*; (2)* $\sum_t |A_t|, \sum_t |B_t| = O(n^{4/3})$*; (3)* $|\Pi| = O(n^{4/3}/\log^4 \log n)$*; (4)* $|A'_s|, |B'_s| = O(\log^3 \log n)$*, for each* $A'_s \times B'_s \in \Pi$.

Proof. We first apply Lemma 6.1 with $A = P$, $B = P$, and $r = n^{1/3}/\log n$. This constructs a collection $\Gamma_1 = \{A_t \times B_t \mid A_t, B_t \subseteq P\}$ of $O(n^{4/3}/\log^4 n)$ edge-disjoint complete bipartite graphs in $O(n^{4/3})$ time. The total size of vertex sets of these graphs is $O(n^{4/3})$, i.e., $\sum_t |A_t|, \sum_t |B_t| = O(n^{4/3})$. We also have a collection $\Pi_1 = \{A'_s \times B'_s \mid A'_s, B'_s \subseteq P\}$ of $O(n^{4/3}/\log^4 n)$ edge-disjoint complete bipartite graphs that record uncertain point pairs, with $|A'_s|, |B'_s| = O(\log^3 n)$.

Hence, the number of uncertain pairs of points of P (i.e., we do not know whether their distances are in $(\alpha, \beta]$) is $\sum_s |A'_s| \cdot |B'_s| = O(n^{4/3} \log^2 n)$. To further reduce this number, we apply Lemma 6.1 on every pair (A'_s, B'_s) of Π_1. More specifically, for each pair (A'_s, B'_s) of Π_1, we apply Lemma 6.1 with $A = A'_s$, $B = B'_s$, and $r = \log n/\log\log n$. This computes a collection Γ_s of $O(\log^4 n/\log^4 \log n)$ edge-disjoint complete bipartite graphs in $O(\log^4 n)$ time; the total size of vertex sets of all graphs in Γ_s is $O(\log^4 n)$. We also have a collection Π_s of $O(\log^4 n/\log^4 \log n)$ edge-disjoint complete bipartite graphs. The size of each vertex set of each graph of Π_s is bounded by $O(\log^3 \log n)$. The total time for Lemma 6.1 on all pairs (A'_s, B'_s) of Π_1 as above is $O(n^{4/3})$. We return $\Gamma_1 \cup \bigcup_s \Gamma_s$ as collection Γ, and $\bigcup_s \Pi_s$ as collection Π in the lemma statement. As such, the complexities in the lemma statement hold. □

In what follows, we describe our algorithm for computing δ^*. Like Katz and Sharir's algorithm [30], our algorithm proceeds in stages. Initially, we have $I_0 = (0, +\infty]$. In each j-th stage, an interval $I_j = (\alpha_i, \beta_j]$ is computed from I_{j-1} such that I_j must contain δ^* and the number of values of $\mathcal{E}(P)$ in I_j is a constant fraction of that in I_{j-1}. Specifically, we will prove that $|\mathcal{E}(P) \cap I_j| = O(n^2\rho^j)$ holds for each j, for some constant $\rho < 1$. Once $|\mathcal{E}(P) \cap I_j|$ is no more than a threshold (to be given later; as will be seen later, this threshold is not constant, which is a main difference between our algorithm and Katz and Sharir's

algorithm [30]), we will compute δ^* directly. In the following we discuss the j-th stage of the algorithm. We assume that we have an interval $I_{j-1} = (\alpha_{j-1}, \beta_{j-1}]$ containing δ^*.

We first apply Lemma 6.2 with $(\alpha, \beta] = (\alpha_{j-1}, \beta_{j-1}]$. This is another major difference between our algorithm and Katz and Sharir's algorithm [30], where they solved the complete BRS problem, while we only solve a partial problem (this saves time by a logarithmic factor). Applying Lemma 6.2 produces a collection $\Gamma_{j-1} = \{A_t \times B_t \mid A_t, B_t \subseteq P\}$ of $O(n^{4/3}/\log^4 \log n)$ edge-disjoint complete bipartite graphs, with $\sum_t |A_t|, \sum_t |B_t| = O(n^{4/3})$, as well as another collection Π_{j-1} of $O(n^{4/3}/\log^4 \log n)$ graphs. By Lemma 6.2 (3) and (4), the number of pairs of points of P in Π_{j-1} is $O(n^{4/3} \log^2 \log n)$.

If $\sum_t |A_t| \cdot |B_t| \leq n^{4/3} \log n$, which is our threshold, then this is the last stage of the algorithm and we compute δ^* directly by the following Lemma 6.3. Each edge of the graph in $\Gamma_{j-1} \cup \Pi_{j-1}$ connects two points of P; we say that the distance of the two points is *induced* by the edge.

Lemma 6.3. *If $\sum_t |A_t| \cdot |B_t| \leq n^{4/3} \log n$, then δ^* can be computed in $O(n^{4/3} \log n)$ time.*

Proof. We first explicitly compute the set S of distances induced from edges of all graphs of Γ_{j-1} and Π_{j-1}. Since $\sum_t |A_t| \cdot |B_t| \leq n^{4/3} \log n$ and the number of edges of all graphs of Π_{j-1} is $O(n^{4/3} \log^2 \log n)$, we have $|S| = O(n^{4/3} \log n)$ and S can be computed in $O(n^{4/3} \log n)$ time by brute force.

Then, we compute the number $k_{\alpha_{j-1}}$ of values of $\mathcal{E}(P)$ that are at most α_{j-1}, which can be done in $O(n^{4/3})$ time [33]. Observe that δ^* is the $(k - k_{\alpha_{j-1}})$-th smallest value in S. Hence, using the linear time selection algorithm, we can find δ^* in $O(|S|)$ time, which is $O(n^{4/3} \log n)$. □

We now assume that $\sum_t |A_t| \cdot |B_t| > n^{4/3} \log n$. The rest of the algorithm for the j-th iteration takes $O(n^{4/3})$ time. For each graph $A_t \times B_t \in \Gamma_{j-1}$, if $|A_t| < |B_t|$, then we switch the name of A_t and B_t, i.e., A_t now refers to B_t and B_t refers to the original A_t. Note that this does not change the solution of the partial BRS produced by Lemma 6.2 and it does not change the complexities of Lemma 6.2 either. This name change is only for ease of the

exposition. Now we have $|A_t| \geq |B_t|$ for each graph $A_t \times B_t \in \Gamma_{j-1}$. Let $m_t = |A_t|$ and $n_t = |B_t|$.

We partition each A_t into $g = \lfloor m_t/n_t \rfloor$ subsets $A_{t1}, A_{t2}, \ldots, A_{tg}$ so that each subset contains n_t elements except that the last subset A_{tg} contains at least n_t but at most $2n_t - 1$ elements. Each pair (A_{ti}, B_t), $1 \leq i \leq g$, can be viewed as a complete bipartite graph. We construct a *d-regular LPS-expander graph* G_{ti} on the vertex set $A_{ti} \cup B_t$ [30, 99], for a constant d to be fixed later. The expander G_{ti} has $O(|A_{ti}|+|B_t|)$ edges and can be computed in $O(|A_{ti}|+|B_t|)$ time [30, 99]. Let G_t be the union of all these expander graphs G_{ti} over all $i = 1, 2, \ldots, g$. The construction of G_t takes $\sum_{i=1}^{g} O(|A_{ti}| + |B_t|) = O(|A_t| + \lfloor \frac{m_t}{n_t} \rfloor \cdot |B_t|) = O(|A_t|)$ time. Hence, computing all graphs $\{G_t\}_t$ for all $O(n^{4/3}/\log^4 \log n)$ pairs $A_t \times B_t$ in Γ_{j-1} takes $\sum_t O(|A_t|) = O(n^{4/3})$ time. The number of edges in G_t is $O(|A_t| + |B_t|)$, and thus the number of edges in all graphs $\{G_t\}_t$ is $\sum_t O(|A_t| + |B_t|) = O(n^{4/3})$.

For each edge (a, b) in graph G_t that connects a point $a \in A_t$ and a point $b \in B_t$, we associate it with the interpoint distance $\|ab\|$. We compute all these distances for all graphs $\{G_t\}_t$ to form a set S. The size of S is bounded by the number of edges in all graphs $\{G_t\}_t$, which is $O(n^{4/3})$. Note that all values of S are in the interval I_{j-1}.

One way we could proceed from here is to find the largest value δ_1 of S with $\delta_1 < \delta^*$ and the smallest value δ_2 with $\delta^* \leq \delta_2$, and then return $(\delta_1, \delta_2]$ as the interval I_j and finish the j-th stage of the algorithm. Finding δ_1 and δ_2 could be done by binary search on S using the linear time selection algorithm and the $O(n^{4/3})$ time decision algorithm. Then the runtime of this step would be $O(n^{4/3} \log n)$, resulting in a total of $O(n^{4/3} \log^2 n)$ time for the overall algorithm for computing δ^* since there are $O(\log n)$ stages. To improve the time, as in [30], we use the "Cole-like" technique to reduce the number of calls to the decision algorithm to $O(1)$ in each stage, as follows.

We assign a *weight* to each value of S. Note that since each graph $G_{ti} \in G_t$ is a d-regular LPS-expander, the degree of G_{ti} is d [30]. Hence, G_{ti} has at most $(|A_{ti}| + |B_t|) \cdot d/2$ edges and thus it contributes at most $(|A_{ti}|+|B_t|) \cdot d/2$ values to S. We assign each distance induced from G_{ti} a weight equal to $|A_{ti}| \cdot |B_t|/(|A_{ti}| + |B_t|)$. As such, the total weight of

the values of S is at most

$$\sum_{t,i}(|A_{ti}| + |B_t|) \cdot \frac{d}{2} \cdot \frac{|A_{ti}| \cdot |B_t|}{|A_{ti}| + |B_t|} = \frac{d}{2} \cdot \sum_{t,i} |A_{ti}| \cdot |B_t| = \frac{d}{2} \cdot m_{j-1},$$

where $m_{j-1} = \sum_t |A_t| \cdot |B_t|$. Recall that $m_{j-1} > n^{4/3} \log n$ and $|B_t| \leq |A_{ti}|$ in each G_{ti}. We can assume $n \geq 16$ so that $m_{j-1} \geq 16$. As such, we have the following bound for the weight of each value in S: $|A_{ti}| \cdot |B_t|/(|A_{ti}| + |B_t|) \leq |B_t| \leq \sqrt{|B_t| \cdot |A_{ti}|} \leq \sqrt{m_{j-1}} \leq m_{j-1}/4$.

We partition the values of S into at most $2d$ intervals $\{I'_1, I'_2, ..., I'_h\}$, $1 \leq h \leq 2d$, such that the total weight of values in every interval is at least $m_{j-1}/4$ and but at most $m_{j-1}/2$. The partition can be done in $O(|S|)$ time, which is $O(n^{4/3})$, using the linear time selection algorithm. Then, we invoke the decision algorithm $\log(2d) = O(1)$ times to find the interval I'_l that contains δ^*, for some $1 \leq l \leq h$. We set $I_j = I'_l$. Since the decision algorithm is called $O(1)$ times, this step takes $O(n^{4/3})$ time. This finishes the j-th stage of the algorithm.

The following Lemma 6.4 shows that the number of values of $\mathcal{E}(P)$ in I_j is a constant portion of that in I_{j-1}. This guarantees that the algorithm will finish in $O(\log n)$ stages since $|\mathcal{E}(P)| = O(n^2)$. As each stage runs in $O(n^{4/3})$ time (except that the last stage takes $O(n^{4/3} \log n)$ time), the total time of the algorithm is $O(n^{4/3} \log n)$.

Lemma 6.4. *There exists a constant ρ with $0 < \rho < 1$ such that the number of values of $\mathcal{E}(P)$ in I_j is at most ρ times the number of values of $\mathcal{E}(P)$ in I_{j-1}.*

Proof. Define n_j (resp., n_{j-1}) as the number of values of $\mathcal{E}(P)$ in I_j (resp., I_{j-1}). Our goal is to find a constant $\rho \in (0, 1)$ so that $n_j \leq \rho \cdot n_{j-1}$ holds.

Recall that m_{j-1} is the number of distances induced from the graphs of Γ_{j-1}. Define m'_{j-1} as the number of distances induced from the graphs of Π_{j-1}. Define q_j (resp., q'_j) as the number of interpoint distances of $\mathcal{E}(P) \cap I_j$ whose point pairs are recorded in Γ_{j-1} (resp., Π_{j-1}). Note that all interpoint distances induced from graphs of Γ_{j-1} are in I_{j-1}. Hence, $m_{j-1} \leq n_{j-1}$. By definition, $n_j = q_j + q'_j$ and $q'_j \leq m'_{j-1}$. By Lemma 6.2 (3) and (4), we have $m'_{j-1} = O(n^{4/3} \log^2 \log n)$.

We make the following **claim:** there exists a constant $\gamma \in (0, 1/3)$ such that $q_j \leq \gamma \cdot m_{j-1}$. Before proving the claim, we prove the lemma using the claim.

As this is not the last stage of the algorithm (since otherwise δ^* would have already been computed without producing interval I_j), it holds that $m_{j-1} > n^{4/3} \log n$. Since $m'_{j-1} = O(n^{4/3} \log^2 \log n)$, there exists a constant $c' \in (0, 1/3)$ such that $\frac{m'_{j-1}}{m_{j-1}} \leq c'$ when n is sufficiently large. As $n_j = q_j + q'_j$, $q'_j \leq m'_{j-1}$, and $m_{j-1} \leq n_{j-1}$, we can obtain the following using the above claim:

$$n_j = q_j + q'_j \leq q_j + m'_{j-1} \leq \gamma \cdot m_{j-1} + c' \cdot m_{j-1} \leq (\gamma + c') \cdot m_{j-1} \leq (\gamma + c') \cdot n_{j-1}.$$

Set $\rho = \gamma + c'$. Since both γ and c' are in $(0, 1/3)$, we have $\rho \in (0, 2/3)$ and $n_j \leq \rho \cdot n_{j-1}$. This proves the lemma.

Proof of the claim. We now prove that there exists a constant $\gamma \in (0, 1/3)$ such that $q_j \leq \gamma \cdot m_{j-1}$. The proof is similar to that in [30].

Consider a pair (A_{ti}, B_t), $1 \leq i \leq g$, obtained in our algorithm. Some edges of the graph G_{ti} induce interpoint distances in S, which may be in I_j. We partition all such graphs G_{ti} into two sets. Let $\mathcal{G}_1$ denote the set of those graphs G_{ti} that contribute fewer than $3(|A_{ti}| + |B_t|)$ interpoint distances in $S \cap I_j$, and $\mathcal{G}_2$ the set of the rest of such graphs (each of them contributes at least $3(|A_{ti}| + |B_t|)$ interpoint distances in $S \cap I_j$).

The set $\mathcal{G}_1$. We first consider set $\mathcal{G}_1$. For a graph $G_{ti} \in \mathcal{G}_1$ built on pair (A_{ti}, B_t), let $\mathcal{D}_{ti}$ be the set of annuli centered at points of A_{ti} with radii α_j and β_j (recall that $I_j = (\alpha_j, \beta_j]$). For the purpose of analysis only, we construct a $1/r$-cutting Ξ for the boundary circles of the annuli in $\mathcal{D}_{ti}$, where r is a constant to be specified later. This partitions the plane into $O(r^2)$ cells such that each cell intersects at most $O(|\mathcal{D}_{ti}|/r)$ boundary circles of annuli in $\mathcal{D}_{ti}$.

For each cell $\sigma \in \Xi$, let $B_t(\sigma)$ denote the set of points of B_t inside σ, $\mathcal{D}_{ti}(\sigma)$ the set of annuli of $\mathcal{D}_{ti}$ that fully contains σ, and $\mathcal{D}'_{ti}(\sigma)$ the set of annuli of $\mathcal{D}_{ti}$ that have at least one

boundary circle intersecting σ. Let N_{ti} denote the number of interpoint distances between points of A_{ti} and points of B_t that are in I_j. Then we have

$$N_{ti} \leq \sum_{\sigma \in \Xi} |\mathcal{D}_{ti}(\sigma)| \cdot |B_t(\sigma)| + \sum_{\sigma \in \Xi} |\mathcal{D}'_{ti}(\sigma)| \cdot |B_t(\sigma)|$$

Since the number of annuli of $\mathcal{D}_{ti}$ that intersect a cell $\sigma \in \Xi$ is $O(|\mathcal{D}_{ti}|/r)$ and $|\mathcal{D}_{ti}| = |A_{ti}|$, we have $|\mathcal{D}'_{ti}(\sigma)| = O(|A_{ti}|/r)$. Using $\sum_{\sigma \in \Xi} |B_t(\sigma)| = |B_t|$, we can derive

$$\sum_{\sigma \in \Xi} |\mathcal{D}'_{ti}(\sigma)| \cdot |B_t(\sigma)| = O\left(\frac{|A_{ti}| \cdot |B_t|}{r}\right)$$

Now we consider $\sum_{\sigma \in \Xi} |\mathcal{D}_{ti}(\sigma)| \cdot |B_t(\sigma)|$. Let $A_{ti}(\sigma) \subseteq A_{ti}$ denote the set of centers of the annuli of $\mathcal{D}_{ti}(\sigma)$. For any point $a \in A_{ti}(\sigma)$ and $b \in B_t(\sigma)$, their distance $\|ab\|$ is in I_j by the definition of $\mathcal{D}_{ti}(\sigma)$. If an edge connecting a and b exists in graph G_{ti}, then $\|ab\|$ must be in S and thus is in I_j as well, i.e., such an edge of G_{ti} contributes a value in $S \cap I_j$. Since G_{ti} is in $\mathcal{G}_1$, it has fewer than $3(|A_{ti}| + |B_t|)$ edges whose induced interpoint distances are in I_j, which implies that the number of edges of G_{ti} connecting points of $A_{ti}(\sigma)$ and points of $B_t(\sigma)$ in G_{ti} is smaller than $3(|A_{ti}| + |B_t|)$. According to Corollary 2.5 in [30], if X and Y are two vertex subsets of a d-regular expander graph of M vertices and there are fewer than $3M$ edges connecting points of X and points of Y, then $|X| \cdot |Y| \leq 9M^2/d$. Applying this result (with $X = A_{ti}(\sigma)$, $Y = B_t(\sigma)$, and $M = |A_{ti}| + |B_t|$), we can derive the following

$$\sum_{\sigma \in \Xi} |\mathcal{D}_{ti}(\sigma)| \cdot |B_t(\sigma)| \leq O(r^2) \cdot \frac{9(|A_{ti}| + |B_t|)^2}{d} = O\left(\frac{r^2(|A_{ti}| + |B_t|)^2}{d}\right)$$

In summary, we have,

$$N_{ti} = O\left(\frac{|A_{ti}| \cdot |B_t|}{r}\right) + O\left(\frac{r^2(|A_{ti}| + |B_t|)^2}{d}\right).$$

Since $|B_t| \leq |A_{ti}| \leq 2|B_t|$ by our partition of set A_t, we have $(|A_{ti}| + |B_t|)^2 \leq 5|A_{ti}| \cdot |B_t|$, which leads to

$$N_{ti} = O\left(\left[\frac{1}{r} + \frac{r^2}{d}\right] \cdot |A_{ti}| \cdot |B_t|\right)$$

By setting $r = d^{1/3}$ and c to be appropriately proportional to $1/d^{1/3}$, we obtain $N_{ti} \leq c \cdot |A_{ti}| \cdot |B_t|$. Summing up all these inequalities for all graphs G_{ti} in set $\mathcal{G}_1$ leads to $N(\mathcal{G}_1) \leq c \cdot \sum_{G_{ti} \in \mathcal{G}_1} |A_{ti}| \cdot |B_t|$, where $N(\mathcal{G}_1)$ is the number of distances between points of A_{ti} and points of B_t that are in I_j for all graphs $G_{ti} \in \mathcal{G}_1$. Since $\sum_{G_{ti} \in \mathcal{G}_1} |A_{ti}| \cdot |B_t| \leq \sum_t |A_t| \cdot |B_t| = m_{j-1}$, we obtain $N(\mathcal{G}_1) \leq c \cdot m_{j-1}$.

The set $\mathcal{G}_2$. We now consider the set $\mathcal{G}_2$. Since each graph $G_{ti} \in \mathcal{G}_2$ contributes at least $3(|A_{ti}| + |B_t|)$ interpoint distances in $S \cap I_j$, G_{ti} contributes at least $3|A_{ti}| \cdot |B_t|$ to the total weight of distances in $S \cap I_j$. Recall that the total weight of distances in $S \cap I_j$ is at most $m_{j-1}/2$ by our algorithm, thus we have $\sum_{G_{ti} \in \mathcal{G}_2} |A_{ti}| \cdot |B_t| \leq m_{j-1}/6$. Let $N(\mathcal{G}_2)$ denote the number of distances between points of A_{ti} and points of B_t that are in I_j for all graphs $G_{ti} \in \mathcal{G}_2$. We have $N(\mathcal{G}_2) \leq \sum_{G_{ti} \in \mathcal{G}_2} |A_{ti}| \cdot |B_t|$ since $I_j \subseteq I_{j-1}$. Therefore, $N(\mathcal{G}_2) \leq m_{j-1}/6$.

Summary. By definition, $q_j = N(\mathcal{G}_1) + N(\mathcal{G}_2)$. As $N(\mathcal{G}_1) \leq c \cdot m_{j-1}$ and $N(\mathcal{G}_2) \leq m_{j-1}/6$, we can derive

$$q_j = N(\mathcal{G}_1) + N(\mathcal{G}_2) \leq c \cdot m_{j-1} + \frac{1}{6} \cdot m_{j-1} = (c + \frac{1}{6}) \cdot m_{j-1}.$$

Let $\gamma = c + 1/6$. Then $\gamma < 1/3$ if d is sufficiently large. As such, we have $q_j \leq \gamma \cdot m_{j-1}$ for a constant $\gamma \in (0, 1/3)$. The claim is thus proved. □

We conclude with the following result.

Theorem 6.2. *Given a set P of n points in the plane and an integer $1 \leq k \leq \binom{n}{2}$, the k-th smallest interpoint distance of P can be computed in $O(n^{4/3} \log n)$ time.*

Note that once δ^* is computed, one can find a pair of points of P whose distance is equal to δ^* in additional $O(n^{4/3})$ time [33].

A bipartite version. Our algorithm can be easily extended to the following *bipartite version* of the distance selection problem: Given a set A of m points and a set B of n points in the plane, and an integer $1 \leq k \leq mn$, compute the k-th smallest interpoint distance δ^* in the set $\{\|ab\| \mid a \in A, b \in B\}$. The decision problem can be solved in $O(m^{2/3}n^{2/3}+m\log n+n\log m)$ time [33]. To adapt our algorithm to compute δ^*, each stage of the algorithm still computes an interval I_j as before. In the j-th stage, we solve the partial BRS problem for A and B with respect to the interval I_{j-1}. We can obtain a result similar to Lemma 6.2 (by using Lemma 6.2 as a subroutine in an analogous way to Theorem 6.1 for dealing with the asymmetric case). More specifically, if $m \leq n < m^2$ (resp. $n \leq m < n^2$), we construct a hierarchical cutting and process those unsolved subproblems by applying Lemma 6.2 with $r = n/m$ (resp. $r = m/n$). If $n \geq m^2$ or $m \geq n^2$, we construct a hierarchical cutting and process those unsolved subproblems in a straightforward manner. As such, we can obtain a collection Γ of $O(m^{2/3}n^{2/3}/\log^4\log(m^2/n)+m^{2/3}n^{2/3}/\log^4\log(n^2/m)+m+n)$ edge-disjoint complete bipartite graphs that record some pairs of $A \times B$ whose interpoint distances are in I_{j-1}. The total size of vertex sets of all graphs in Γ is $O(m^{2/3}n^{2/3}+m\log n+n\log m)$. We also have another collection Π of edge-disjoint complete bipartite graphs that record a total of $O(m^{2/3}n^{2/3}\log^2\log(m+n))$ uncertain pairs of $A \times B$, i.e., we do not know whether their distances are in I_{j-1}. The total runtime is $O(m^{2/3}n^{2/3}+m\log n+n\log m)$. We compute the number of interpoint distances induced from collection Γ. If this number is at most $(m^{2/3}n^{2/3}+m\log n+n\log m)\log(m+n)$, then this is the last stage of the algorithm and we compute δ^* directly. Otherwise, we use the "Cole-like" technique to perform a binary search on the interpoint distances induced from the expander graphs that are built on vertex sets of the graphs in Γ, which calls the decision algorithm $O(1)$ times. The algorithm will finish within $O(\log(m+n))$ stages by similar analysis to Lemma 6.4. As such, the bipartite distance selection problem can be solved in $O((m^{2/3}n^{2/3}+m\log n+n\log m)\log(m+n))$ time.

6.4 Two-sided discrete Fréchet distance with shortcuts

In this section, we show that our techniques in Section 6.3 can be used to solve the two-sided DFD problem. Let $A = \{a_1, a_2, ..., a_m\}$ and $B = \{b_1, b_2, ..., b_n\}$ be two sequences of points in the plane. Consider two frogs connected by an inelastic leash, initially placed at a_1 and b_1, respectively. Each frog is allowed to jump forward at most one step in one move, i.e., if the first frog is currently at a_i, then in the next move it can either jump to a_{i+1} or stay at a_i. Note that frogs are not allowed to go backwards. The *discrete Fréchet distance* (or DFD for short) is defined as the minimum length of the inelastic leash that allows two frogs to reach their destinations, i.e., a_m and b_n, respectively.

Because the Fréchet distance is very sensitive to outliers, to reduce the sensitivity, DFD with outliers have been proposed [36]. Specifically, if we allow the A-frog to jump from its current point to any of its succeeding points in each move but B-frog has to traverse all points in B in order plus one restriction that only one frog is allowed to jump in each move (i.e., in each move one of the frogs must stay still), then this problem is called *one-sided discrete Fréchet distance with shortcuts* (or *one-sided DFD* for short), where the goal is to compute the minimum length of the inelastic leash that allows two frogs to reach their destinations. If we allow both frogs to skip points in their sequences (but again with the restriction that only one frog is allowed to jump in each move), then problem is called *two-sided DFD*.

We focus on the two-sided DFD in this section while the one-sided version will be treated in the next section. Let δ^* denote the optimal objective value, i.e., the minimum length of the leash. Avraham, Filtser, Kaplan, Katz, and Sharir [36] presented an algorithm that can compute δ^* in $O((m^{2/3}n^{2/3}+m+n)\log^3(m+n))$ time. In what follows, we show that our techniques in Section 6.3 can improve their algorithm to $O((m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m\log n + n\log m)\log(m+n))$ time, roughly a factor of $O(\log^2(m+n))$ faster.

To solve the problem, the authors of [36] first proposed an algorithm to solve the decision problem, i.e., given any δ, decide whether $\delta^* \leq \delta$; the algorithm runs in $O((m^{2/3}n^{2/3} + m + n)\log^2(m+n))$ time. Then, to compute δ^*, the authors of [36] used the bipartite

version of the distance selection algorithm from Katz and Sharir [30] for point sets A and B together with their decision algorithm to do binary search on the interpoint distances between points in A and those in B, i.e., in each iteration, using the distance selection algorithm to find the k-th smallest distance δ_k for an appropriate k and then call the decision algorithm on δ_k to decide which way to search. As both the distance selection algorithm [30] and the decision algorithm run in $O((m^{2/3}n^{2/3} + m + n)\log^2(m+n))$ time, computing δ^* takes $O((m^{2/3}n^{2/3} + m + n)\log^3(m+n))$ time.

In what follows, we first show that the runtime of their decision algorithm can be reduced by a factor of roughly $O(\log^2(m+n))$ using our result in Theorem 6.1 for the complete BRS problem, and then discuss how to improve the optimization algorithm for computing δ^*.

Improving the decision algorithm. The basic idea of the decision algorithm in [36] is to consider a matrix M whose rows and columns correspond to points in sequences A and B, respectively. Each entry $M(i,j)$ of M is 1 if $\|a_i b_j\| \leq \delta$, and 0 otherwise. One can determine whether there exists a path from $M(1,1)$ to $M(m,n)$ in M that only consists of value 1 by performing "upward" and "rightward" moves. The matrix M is not computed explicitly. The algorithm first performs a complete BRS with $\alpha = 0$ and $\beta = \delta$ using a result from [30] on A and B, which generates a collection $\Gamma = \{A_t \times B_t\}_t$ of complete bipartite graphs that record all pairs of $A \times B$ whose interpoint distances are at most δ in $O((m^{2/3}n^{2/3} + m + n)\log(m+n))$ time, with $\sum_t |A_t|, \sum_t |B_t| = O((m^{2/3}n^{2/3} + m + n)\log(m+n))$. Each edge of these graphs corresponds to an entry of value 1 in M. Then for each graph $A_t \times B_t \in \Gamma$, points of A_t and B_t are sorted by their index order into lists $\mathcal{L}_{A_t}$ and $\mathcal{L}_{B_t}$, respectively. The sorting takes $O((m^{2/3}n^{2/3} + m + n)\log^2(m+n))$ time in total. With these information in hand, the rest of the algorithm runs in time linear in the total size of vertex sets of graphs in Γ, which is $O((m^{2/3}n^{2/3} + m + n)\log(m+n))$.

We can improve their decision algorithm by applying our complete BRS result in Theorem 6.1. Specifically, applying Theorem 6.1 will produce in $O(m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m\log n + n\log m)$ time a collection Γ of complete bipartite graphs that record all pairs of

$A \times B$ whose interpoint distances are at most δ. To reduce the time on the sorting step, when computing the canonical subsets $B(\sigma)$ in Lemma 6.1, we process points of B following their index order. Similarly, when computing the canonical sets of A_σ, we process the circles of $\mathcal{C}_{\sigma'}$ following the index order of their centers in A. This ensures that points in each A_t and each B_t are sorted automatically during the construction, i.e., lists $\mathcal{L}_{A_t}$ and $\mathcal{L}_{B_t}$ are available once the algorithm of Theorem 6.1 is done. The rest of the algorithm follows exactly the same as the algorithm in [36], which takes time proportional to the total size of vertex sets of graphs in Γ, i.e., $O(m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m \log n + n \log m)$ by Theorem 6.1. As such, the total time of the new decision algorithm is $O(m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m \log n + n \log m)$.

Improving the optimization algorithm. With our new $O((m^{2/3}n^{2/3} + m \log n + n \log m) \log(m+n))$ time bipartite distance selection algorithm in Section 6.3 and the above faster decision algorithm, following the same binary search scheme as discussed above, δ^* can be computed in $O((m^{2/3}n^{2/3} + m \log n + n \log m) \log^2(m+n))$ time, a logarithmic factor improvement over the result of [36]. Notice that the time is dominated by the calls to the bipartite distance selection algorithm.

To further improve the algorithm, an observation is that we do not have to call the distance selection algorithm as an oracle and instead we can use that algorithm as a framework and replace the decision algorithm of the distance selection problem by the decision algorithm of the two-sided DFD problem. This will roughly reduce another logarithmic factor. The proof of the following theorem provides the details about this idea.

Theorem 6.3. *Given two sequences of points $A = (a_1, a_2, ..., a_m)$ and $B = (b_1, b_2, ..., b_n)$ in the plane, the two-sided DFD problem can be solved in $O((m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m \log n + n \log m) \log(m+n))$ time.*

Proof. Following our distance selection algorithm, we run in stages and each j-th stage will compute an interval I_j that contains δ^*. In the j-th stage, we first perform the partial BRS on point sets A and B with respect to interval I_{j-1}, in the same way as before. This produces a collection Γ of $(m^{2/3}n^{2/3}/\log^4 \log(m^2/n) + m^{2/3}n^{2/3}/\log^4 \log(n^2/m) + m + n)$

edge-disjoint complete bipartite graphs that record some pairs of $A \times B$ whose interpoint distances are in I_{j-1}. The total size of vertex sets of all graphs in Γ is $O(m^{2/3}n^{2/3} + m\log n + n\log m)$. In addition, we also have a collection Π of complete bipartite graphs representing $O(m^{2/3}n^{2/3}\log^2\log(m+n))$ uncertain pairs of $A \times B$. The total runtime is $O(m^{2/3}n^{2/3} + m\log n + n\log m)$.

We next compute the number n_Γ of distances induced from the graphs of Γ. If n_Γ is larger than the threshold $\tau = (m^{2/3}n^{2/3} + m\log n + n\log m)\log(m+n)$, then we use the "Cole-like" technique to perform a binary search on the interpoint distances induced from the expander graphs that are built on the vertex sets of the graphs in Γ, which calls the decision algorithm $O(1)$ times. The runtime for this stage is $O(m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m\log n + n\log m)$. If $n_\Gamma \leq \tau$, then we reach the last stage of the algorithm and we can compute δ^* as follows. We compute the interpoint distances induced from the graphs in Γ and Π. The total number of such distances is $O((m^{2/3}n^{2/3} + m\log n + n\log m)\log(m+n))$. Using the decision algorithm and the linear time selection algorithm, a binary search on these interpoint distances is performed to compute δ^*, which takes $O((m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m\log n + n\log m)\log(m+n))$ time as the decision algorithm is called $O(\log(m+n))$ times. The algorithm finishes within $O(\log(m+n))$ stages by an analysis similar to Lemma 6.4 (indeed, the proof of Lemma 6.4 does not rely on which decision algorithm is used).

In summary, the total runtime for computing δ^* is bounded by $O((m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))} + m\log n + n\log m)\log(m+n))$. □

A general (deterministic) algorithmic framework. The algorithm of Theorem 6.3 can be made into a general algorithmic framework for solving geometric optimization problems involving interpoint distances in the plane. Specifically, suppose we have an optimization problem $\mathcal{P}$ whose optimal objective value δ^* is equal to $\|ab\|$ for a point $a \in A$ and a point $b \in B$, with A as a set of m points and B as a set of n points in the plane. The goal is to compute δ^*. Suppose that we have a decision algorithm that can determine whether $\delta \geq \delta^*$ in T_D time for any δ. Then, we can compute δ^* by applying exactly the same algorithm of Theorem 6.3 except that we use the decision algorithm for $\mathcal{P}$ instead. The total time of

the algorithm is $O((m^{2/3}n^{2/3} + m\log n + n\log m + T_D)\cdot\log(m+n))$. Note that in the case $T_D = o((m^{2/3}n^{2/3} + m\log n + n\log m)\log(m+n))$ this is faster than the traditional binary search approach by repeatedly invoking the distance selection algorithm.

Theorem 6.4. *Given two sets A and B of m and n points respectively in the plane, any geometric optimization problem whose optimal objective value is equal to the distance between a point of $a \in A$ and a point of $b \in B$ can be solved in $O((m^{2/3}n^{2/3} + m\log n + n\log m + T_D)\cdot\log(m+n))$ time, where T_D is the time for solving the decision version of the problem.*

6.5 One-sided discrete Fréchet distance with shortcuts

In this section, we consider the one-sided DFD problem, defined in Section 6.4. Let δ^* denote the optimal objective value. Avraham, Filtser, Kaplan, Katz, and Sharir [36] proposed an a randomized algorithm of $O((m+n)^{6/5+\epsilon})$ expected time. We show that using our result in Lemma 6.1 for the partial BRS problem the runtime of their algorithm can be reduced to $O((m+n)^{6/5}\log^{8/5}(m+n))$.

Define $\mathcal{E}(A,B) = \{\|ab\| \mid a \in A, b \in B\}$. It is known that $\delta^* \in \mathcal{E}(A,B)$ [36]. The decision problem is to decide whether $\delta \geq \delta^*$ for any δ. The authors [36] first solved the decision problem in $O(m+n)$ (deterministic) time. To compute δ^*, their algorithm has two main procedures.

The first main procedure computes an interval $(\alpha,\beta]$ that is guaranteed to contain δ^*, and in addition, with high probability the interval contains at most L values of $\mathcal{E}(A,B)$, given any $1 \leq L \leq mn$; the algorithm runs in $O((m+n)^{4/3+\epsilon}/L^{1/3} + (m+n)\log(m+n)\log\log(m+n))$ time, for any $\epsilon > 0$. More specifically, during the course of the algorithm, an interval $(\alpha,\beta]$ containing δ^* is maintained; initially $\alpha = 0$ and $\beta = \infty$. In each iteration, the algorithm first determines, through random sampling, whether the number of values of $\mathcal{E}(A,B)$ in $(\alpha,\beta]$ is at most L with high probability. If so, the algorithm stops by returning the current interval $(\alpha,\beta]$. Otherwise, a subset R of $O(\log(m+n))$ values of $\mathcal{E}(A,B)$ is sampled which contains with high probability an approximate median (in the middle three quarters) among the values of $\mathcal{E}(A,B)$ in $(\alpha,\beta]$. A binary search guided by the decision

algorithm is performed to narrow down the interval $(\alpha, \beta]$; the algorithm then proceeds with the next iteration. As such, after $O(\log(m+n))$ iterations, the algorithm eventually returns an interval $(\alpha, \beta]$ with the property discussed above.

The second main procedure is to find δ^* from $\mathcal{E}(A,B) \cap (\alpha, \beta]$. This is done by using a *bifurcation tree* technique (Lemma 4.4 [36]), whose runtime relies on L', the true number of values of $\mathcal{E}(A,B)$ in $(\alpha, \beta]$. As it is possible that $L' > L$, if the algorithm detects that case happens, then the first main procedure will run one more round from scratch. As $L' < L$ holds with high probability, the expected number of rounds is $O(1)$. If $L' \leq L$, the runtime of the second main procedure is bounded by $O((m+n)L^{1/2}\log(m+n))$.

As such, the expected time of the algorithm is $O((m+n)^{4/3+\epsilon}/L^{1/3} + (m+n)\log(m+n)\log\log(m+n) + (m+n)L^{1/2}\log(m+n))$. Setting L to $O((m+n)^{2/5+\epsilon})$ for another small $\epsilon > 0$, the time can be bounded by $O((m+n)^{6/5+\epsilon})$.

Our improvement. We can improve the runtime of the first main procedure by a factor of $O((m+n)^{\epsilon})$, which leads to the improvement of overall algorithm by a similar factor. To this end, by applying Lemma 6.1 with $r = (\frac{m+n}{L})^{1/3}$, we first have the following corollary, which improves Lemma 4.1 in [36] (which is needed in the first main procedure).

Corollary 6.5. *Given a set A of m points and a set B of n points in the plane, an interval $(\alpha, \beta]$, and a parameter $1 \leq L \leq mn$, we can compute in $O((m+n)^{4/3}/L^{1/3} \cdot \log(\frac{m+n}{L}))$ time two collections $\Gamma(A,B,\alpha,\beta) = \{A_t \times B_t \mid A_t \subseteq A, B_t \subseteq B\}$ and $\Pi(A,B,\alpha,\beta) = \{A'_s \times B'_s \mid A'_s \subseteq A, B'_s \subseteq B\}$ of edge-disjoint complete bipartite graphs that satisfy the conditions of Problem 6.1, with the following complexities: (1) $|\Gamma| = O((\frac{m+n}{L})^{4/3})$; (2) $\sum_t |A_t|, \sum_t |B_t| = O((m+n)^{4/3}/L^{1/3} \cdot \log(\frac{m+n}{L}))$; (3) $|\Pi| = O((\frac{m+n}{L})^{4/3})$; (4) $|A'_s| = O(\frac{mL}{m+n})$ and $|B'_s| = O(\frac{nL}{m+n})$ for each $A'_s \times B'_s \in \Pi$; (5) the number of pairs of points recorded in Π is $O((m+n)^{4/3}L^{2/3})$.*

Replacing Lemma 4.1 in [36] by our results in Corollary 6.5 and following the rest of the algorithm in [36] leads to an algorithm to compute δ^* in $O((m+n)^{6/5}\log^2(m+n))$ time. To make the chapter more self-contained, we present some details below. Also, we

put the discussion in the context of a more general algorithmic framework (indeed, a recent result of Katz and Sharir [40] already gave such a framework; here we improve their result by a factor of $O((m+n)^{\epsilon})$ due to Corollary 6.5).

A general (randomized) algorithmic framework. Suppose we have an optimization problem $\mathcal{P}$ whose optimal objective value δ^* is equal to $\|ab\|$ for a point $a \in A$ and a point $b \in B$, with A as a set of m points and B as a set of n points in the plane. The goal is to compute δ^*. Suppose that we have a decision algorithm that can determine whether $\delta \geq \delta^*$ in T_D time for any δ. With the result from Corollary 6.5, we have the following lemma. Define $\mathcal{E}(A, B)$ in the same way as above.

Lemma 6.5. *Given any $1 \leq L \leq mn$, there is a randomized algorithm that can compute an interval $(\alpha, \beta]$ that contains δ^* and with high probability contains at most L values of $\mathcal{E}(A, B)$; the expected time of the algorithm is $O((m+n)^{4/3}/L^{1/3} \cdot \log^2(m+n) + T_D \cdot \log(m+n) \cdot \log\log(m+n))$.*

Proof. We maintain an interval $(\alpha, \beta]$ (which is initialized to $(0, +\infty]$) containing δ^* and shrink it iteratively. In each iteration, we first invoke Corollary 6.5 to obtain two collections $\Gamma(A, B, \alpha, \beta)$ and $\Pi(A, B, \alpha, \beta)$ of complete bipartite graphs in $O((m+n)^{4/3}/L^{1/3} \cdot \log(\frac{m+n}{L}))$ time. In particular, the graphs of $\Pi(A, B, \alpha, \beta)$ record uncertain point pairs of $A \times B$ that we do not know whether their distances are in $(\alpha, \beta]$. The total number of these uncertain pairs is $M = O((m+n)^{4/3}L^{2/3})$.

Let S_1 (resp., S_2) denote the set of interpoint distances recorded in collection $\Gamma(A, B, \alpha, \beta)$ (resp., $\Pi(A, B, \alpha, \beta)$). Note that $|S_2| = M$ and all values of S_1 are in $(\alpha, \beta]$ while some values of S_2 may not be in $(\alpha, \beta]$. Define S_2' to be the subset of distances of S_2 that lie in $(\alpha, \beta]$. We need to determine the number of distances of $S_1 \cup S_2$ that lie in $(\alpha, \beta]$, i.e., determine $|S_1| + |S_2'|$. To this end, as $|S_1| = \sum_t |A_t| \cdot |B_t|$ and $\sum_t |A_t|, \sum_t |B_t| = O((m+n)^{4/3}/L^{1/3} \cdot \log(\frac{m+n}{L}))$, $|S_1|$ can be easily computed in $O((m+n)^{4/3}/L^{1/3} \cdot \log(\frac{m+n}{L}))$ time. It remains to determine $|S_2'|$. A method is proposed in Lemma 4.2 of [36] to determine with high probability whether $|S_2'| \leq L/2$. This is done by generating a random

sample R_2 of $c_2(M/L \cdot \log(m+n))$ values from S_2, for a sufficiently large constant $c_2 > 0$, and then check how many of them lie in $(\alpha, \beta]$. The runtime of this step is $O(|R_2|)$, i.e., $O(M/L \cdot \log(m+n)) = O((m+n)^{4/3}/L^{1/3} \cdot \log(m+n))$.

If $|S_1| \leq L/2$ and the above approach determines that $|S_2'| \leq L/2$, then with high probability the total number of distances of $\mathcal{E}(A,B) \cap (\alpha, \beta]$ is at most L and we are done with the lemma. Otherwise, an approach is given in Lemma 4.3 of [36] to generate a sample R of $O(\log(m+n))$ distances from $S_1 \cup S_2$, so that with high probability R contains an approximate median (in the middle three quarters) among the values of $\mathcal{E}(A,B)$ in $(\alpha, \beta]$; this step takes $O((m+n)^{4/3}/L^{1/3} \cdot \log(m+n))$ time.

We now call the decision algorithms to do binary search on the values of R to find two consecutive values α', β' in R such that $\delta^* \in (\alpha', \beta']$. Note that $(\alpha', \beta'] \subseteq (\alpha, \beta]$, and $(\alpha', \beta']$ contains with high probability at most $7/8$ distances of $\mathcal{E}(A,B)$ in $(\alpha, \beta]$. As $|R| = O(\log(m+n))$, we need to call the decision algorithm $O(\log\log(m+n))$ times, and thus computing $(\alpha', \beta']$ takes $O(T_D \cdot \log\log(m+n))$ time. This finishes one iteration of the algorithm, which takes $O((m+n)^{4/3}/L^{1/3} \cdot \log(m+n) + T_D \cdot \log\log(m+n))$ time in total.

We then proceed with the next iteration with $(\alpha, \beta] = (\alpha', \beta']$. The exptected number of iterations of the algorithm is $O(\log(m+n))$. Hence, the expected time of the overall algorithm is $O((m+n)^{4/3}/L^{1/3} \cdot \log^2(m+n) + T_D \cdot \log(m+n) \cdot \log\log(m+n))$. □

With the interval $(\alpha, \beta]$ computed by Lemma 6.5, the next step is to compute δ^* from $\mathcal{E}(A,B) \cap (\alpha, \beta]$. This is done using bifurcation tree technique (Lemma 4.4 [36]) as discussed before; see also Section 2.2 of [40] for a discussion on more general problems. The runtime of this step is $O(T_D \cdot L^{1/2} \cdot \log(m+n))$ (see Proposition 2.6 [40]).

In summary, the total time of the algorithm is $O((m+n)^{4/3}/L^{1/3} \cdot \log^2(m+n) + T_D \cdot \log(m+n) \cdot \log\log(m+n) + T_D \cdot L^{1/2} \cdot \log(m+n))$. We thus have the following theorem.

Theorem 6.6. *Given two sets A and B of m and n points respectively in the plane, any geometric optimization problem whose optimal objective value is equal to the distance between a point of $a \in A$ and a point of $b \in B$ can be solved by a randomized algorithm of*

$O((m+n)^{4/3}/L^{1/3} \cdot \log^2(m+n) + T_D \cdot \log(m+n) \cdot \log\log(m+n) + T_D \cdot L^{1/2} \cdot \log(m+n))$ *expected time, for any parameter* $1 \leq L \leq mn$.

For the one-sided DFD problem, we have $T_D = O(m+n)$. Setting $L = (m+n)^{2/5} \log^{6/5}(m+n)$ leads to the following result.

Corollary 6.7. *Given a sequence A of m points and another sequence B of n points in the plane, the one-sided discrete Fréchet distance with shortcuts problem can be solved by a randomized algorithm of $O((m+n)^{6/5} \log^{8/5}(m+n))$ expected time.*

As discussed in Section 6.1, another immediate application of Theorem 6.6 is the reverse shortest path problem in unit-disk graphs.

CHAPTER 7

FUTURE WORK

In this chapter, we discuss several future works that are natural extensions of the problems we studied before.

7.1 Euclidean minimum moving spanning tree

The *Euclidean minimum moving spanning tree* (moving-EMST for short) for a set of moving points can be defined in a similar way to the definition of the Euclidean minimum bottleneck moving spanning tree as we discussed in Chapter 4. It has been proved that the problem of computing a moving-EMST is NP-hard by a reduction from the Partition problem. An $O(n^2)$-time 2-approximation algorithm was proposed by computing the EMST of a complete graph defined on the moving points [20]. Note that such an approximation ratio is tight. Moreover, the moving-EMST is equivalent to the minimum spanning tree of a point set in $\mathbb{R}^4$ with a non-Euclidean metric. An $O(n \log n)$-time $(2+\epsilon)$-approximation algorithm was presented [20] by following this observation.

I plan to study approximation algorithms for the moving-EMST problem. Considering the potential connections between the moving spanning tree and unit-disk graphs, I expect that the underlying geometric properties of unit-disk graphs may help us discover more essential observations and develop more efficient approximation algorithms for this problem further.

7.2 Reverse shortest path problem in unit-ball graphs

Similar to the definition of unit-disk graphs as we discussed earlier, the unit-ball graph can be defined in 3D space. Specifically, given a set P of n points in 3-dimensional space and a parameter $r > 0$, a unit-ball graph $G(P)$ uses P as its vertex set, and any two points of P are connected if their distance is no larger than r. In the reverse shortest path problem

for unit-ball graphs, we are given two points $s, t \in P$ and another parameter $\lambda > 0$, and the target is to compute the smallest r such that the distance between s and t is at most λ. This problem has a preliminary randomized solution of $O(n^{17/12+\epsilon})$, $\epsilon > 0$ expected running time in the literature [40]. This solution is based on a framework that adopts semi-algebraic range searching and bifurcation-tree techniques.

I plan to exploit additional properties of 3D unit-ball graphs and adapt the methodology (parametric search) of our solution for the 2D case to this 3D case. The critical part is choosing a decision algorithm to parameterize. One decision algorithm (single-source shortest path algorithm) for unit-ball graphs was presented in the previous work which extends the solution in the 2D case to the 3D case. This involves a modified version of the problem of "unit-ball range searching", i.e., the query points and input points are separated by a plane parallel to xy-plane. This property may allow us to finish the query in a more efficient manner.

7.3 Single-source shortest path problem in weighted unit-disk graphs

This problem was solved in $O(n \log^2 n)$ time [1] and was used as the decision algorithm of our solution to the RSP problem. The previous work has a bottleneck subproblem: 2D offline insertion-only (additively) weighted nearest neighbor (OIWNN). An instance of OIWNN problem with n operations (insertions and queries) can be solved in $O(n \log^2 n)$ time and $O(n)$ space. The SSSP algorithm for unit-disk graphs can be improved immediately if this OIWNN problem can be improved. However, we have another property that may be helpful in the SSSP algorithm for unit-disk graphs, i.e., all query points and insertion points are separated by an axis-parallel line. We have used this property to improve this framework to $O(n \log n)$ running time for the L_1 unit-disk graphs.

I plan to improve the modified OIWNN problem so that the total time complexity of the SSSP algorithm for weighted L_2 unit-disk graphs can be improved to $O(n \log n)$. One direction is to maintain the part of the planar Voronoi diagram above the axis-parallel line dynamically.

REFERENCES

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

[2] J. Dall and M. Christensen, "Random geometric graphs," *Physical review E*, vol. 66, no. 1, p. 016121, 2002.

[3] M. L. Huson and A. Sen, "Broadcast scheduling algorithms for radio networks," in *Proceedings of Military Communications Conference (MILCOM'95)*, vol. 2. IEEE, 1995, pp. 647–651.

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

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

[6] S. Cabello and M. Jejčič, "Shortest paths in intersection graphs of unit disks," *Computational Geometry: Theory and Applications*, vol. 48, no. 4, pp. 360–367, 2015.

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

[8] J. Gao and L. Zhang, "Well-separated pair decomposition for the unit-disk graph metric and its applications," *SIAM Journal on Computing*, vol. 35, no. 1, pp. 151–169, 2005.

[9] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir, "Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications,"

in *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2017, pp. 2495–2504.

[10] L. Roditty and M. Segal, "On bounded leg shortest paths problems," *Algorithmica*, vol. 59, no. 4, pp. 583–600, 2011.

[11] B. N. Clark, C. J. Colbourn, and D. S. Johnson, "Unit disk graphs," *Discrete Mathematics*, vol. 86, pp. 165–177, 1990.

[12] T. 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)*, 1998, pp. 194–200.

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

[14] H. Wang and Y. Zhao, "An optimal algorithm for L_1 shortest paths in unit-disk graphs," in *Proceedings of the 33rd Canadian Conference on Computational Geometry (CCCG)*, 2021, pp. 211–218.

[15] ——, "An optimal algorithm for L_1 shortest paths in unit-disk graphs," *Computational Geometry: Theory and Applications*, vol. 110 (101960), pp. 1–9, 2023.

[16] J. S. Salowe, "L-infinity interdistance selection by parametric search," *Information processing letters*, vol. 30, no. 1, pp. 9–14, 1989.

[17] H. Wang and Y. Zhao, "Reverse shortest path problem for unit-disk graphs," in *Proceedings of the 17th International Symposium of Algorithms and Data Structures (WADS)*, 2021, pp. 655–668.

[18] ——, "Reverse shortest path problem in weighted unit-disk graphs," in *Proceedings of the 16th International Conference and Workshops on Algorithms and Computation (WALCOM)*, 2022, pp. 135–146.

[19] ——, "Reverse shortest path problem for unit-disk graphs," *Journal of Computational Geometry*, vol. 14(1), pp. 14–47, 2023.

[20] H. A. Akitaya, A. Biniaz, P. Bose, J.-L. D. Carufel, A. Maheshwari, L. F. S. X. d. Silveira, and M. Smid, "The minimum moving spanning tree problem," in *Proceedings of the 17th Workshop on Algorithms and Data Structures (WADS)*, 2021, pp. 15–28.

[21] H. Wang and Y. Zhao, "Computing the minimum bottleneck moving spanning tree," in *Proceedings of the 47th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2022, pp. 82:1–82:15.

[22] J. L. Bentley and H. A. Maurer, "A note on Euclidean near neighbor searching in the plane," *Information Processing Letters*, vol. 8, pp. 133–136, 1979.

[23] B. Chazelle, "An improved algorithm for the fixed-radius neighbor problem," *Information Processing Letters*, vol. 16, no. 4, pp. 193–198, 1983.

[24] B. Chazelle, R. Cole, F. P. Preparata, and C.-K. Yap, "New upper bounds for neighbor searching," *Information and control*, vol. 68, no. 1-3, pp. 105–124, 1986.

[25] B. Chazelle and H. Edelsbrunner, "Optimal solutions for a class of point retrieval problems," *Journal of Symbolic Computation*, vol. 1, no. 1, pp. 47–56, 1985.

[26] P. Afshani and T. M. Chan, "Optimal halfspace range reporting in three dimensions," in *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2009, pp. 180–186.

[27] T. M. Chan and K. Tsakalidis, "Optimal deterministic algorithms for 2-d and 3-d shallow cuttings," *Discrete and Computational Geometry*, vol. 56, no. 4, pp. 866–881, 2016.

[28] B. Chazelle and L. J. Guibas, "Fractional cascading: I. A data structuring technique," *Algorithmica*, vol. 1, no. 1, pp. 133–162, 1986.

[29] ——, "Fractional cascading: II. Applieacations," *Algorithmica*, vol. 1, no. 1, pp. 163–191, 1986.

[30] M. J. Katz and M. Sharir, "An expander-based approach to geometric optimization," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1384–1408, 1997.

[31] T. M. Chan and D. W. Zheng, "Hopcroft's problem, log-star shaving, 2D fractional cascading, and decision trees," in *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2022, pp. 190–210, full version with new results available at https://arxiv.org/pdf/2111.03744.pdf.

[32] T. M. Chan, "On enumerating and selecting distances," *International Journal of Computational Geometry and Application*, vol. 11, pp. 291–304, 2001.

[33] H. Wang, "Unit-disk range searching and applications," in *Proceedings of the 18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, 2022, pp. 32:1–32:17.

[34] P. K. Agarwal, R. B. Avraham, H. Kaplan, and M. Sharir, "Computing the discrete Fréchet distance in subquadratic time," *SIAM Journal on Computing*, vol. 43, pp. 429–449, 2014.

[35] H. Alt and M. Godau, "Computing the Fréchet distance between two polygonal curves," *International Journal of Computational Geometry and Applications*, vol. 5, pp. 75–91, 1995.

[36] R. B. Avraham, O. Filtser, H. Kaplan, M. J. Katz, and M. Sharir, "The discrete and semicontinuous Fréchet distance with shortcuts via approximate distance counting and selection," *ACM Transactions on Algorithms*, vol. 11, no. 4, p. Article No. 29, 2015.

[37] K. Buchin, M. Buchin, and Y. Wang, "Exact algorithms for partial curve matching via the Fréchet distance," in *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2009, pp. 645–654.

[38] M. Buchin, A. Driemel, and B. Speckmann, "Computing the Fréchet distance with shortcuts is NP-hard," in *Proceedings of the 30th Annual Symposium on Computational Geometry (SoCG)*, 2014, pp. 367–376.

[39] A. Driemel and S. Har-Peled, "Jaywalking your dog: computing the Fréchet distance with shortcuts," *SIAM Journal on Computing*, vol. 42, no. 5, pp. 1830–1866, 2013.

[40] M. J. Katz and M. Sharir, "Efficient algorithms for optimization problems involving semi-algebraic range searching," *arXiv:2111.02052*, 2021.

[41] J. L. Bentley, "Decomposable searching problems," *Information Processing Letters*, vol. 8, pp. 244–251, 1979.

[42] M. de Berg, K. Buchin, B. Jansen, and G. Woeginger, "Fine-grained complexity analysis of two classic TSP variants," *ACM Transactions on Algorithms*, vol. 17, no. 1, pp. 5:1–5:29, 2021.

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

[44] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, "Optimal point location in a monotone subdivision," *SIAM Journal on Computing*, vol. 15, no. 2, pp. 317–340, 1986.

[45] D. Kirkpatrick, "Optimal search in planar subdivisions," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 28–35, 1983.

[46] P. K. Agarwal, A. Efrat, and M. Sharir, "Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications," *SIAM Journal on Computing*, vol. 29, pp. 912–953, 1999.

[47] C. Liu, "Nearly optimal planar k nearest neighbors queries under general distance functions," in *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020, pp. 2842–2859.

[48] D. Burton and P. L. Toint, "On an instance of the inverse shortest paths problem," *Mathematical Programming*, vol. 53, pp. 45–61, 1992.

[49] J. Zhang and Y. Lin, "Computation of the reverse shortest-path problem," *Journal of Global Optimization*, vol. 25, no. 3, pp. 243–261, 2003.

[50] M. de Berg, H. L. Bodlaender, S. Kisfaludi-Bak, D. Marx, and T. C. van der Zanden, "A framework for ETH-tight algorithms and lower bounds in geometric intersection graphs," in *Proceedings of the 50th Annual ACM Symposium on Theory of Computing (STOC)*, 2018, pp. 574–586.

[51] R. Cole, "Slowing down sorting networks to obtain faster sorting algorithms," *Journal of the ACM*, vol. 34, no. 1, pp. 200–208, 1987.

[52] N. Megiddo, "Applying parallel computation algorithms in the design of serial algorithms," *Journal of the ACM*, vol. 30, no. 4, pp. 852–865, 1983.

[53] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[54] G. Frederickson, "Parametric search and locating supply centers in trees," in *Proc. of the 2nd International Workshop on Algorithms and Data Structures (WADS)*, 1991, pp. 299–319.

[55] G. N. Frederickson and D. B. Johnson, "Generalized selection and ranking: Sorted matrices," *SIAM Journal on Computing*, vol. 13, no. 1, pp. 14–30, 1984.

[56] ——, "Finding kth paths and p-centers by generating and searching good data structures," *Journal of Algorithms*, vol. 4, no. 1, pp. 61–80, 1983.

[57] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, pp. 153–174, 1987.

[58] M. I. Shamos and D. Hoey, "Closest-point problems," in *Proc. of the 16th Annual Symposium on Foundations of Computer Science*, 1975, pp. 151–162.

[59] M. Ajtai, J. Komlós, and E. Szemerédi, "An $O(n \log n)$ sorting network," in *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, 1983, pp. 1–9.

[60] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, vol. 7, pp. 448–461, 1973.

[61] N. Sarnak and R. E. Tarjan, "Planar point location using persistent search trees," *Communications of the ACM*, vol. 29, pp. 669–679, 1986.

[62] J. Erickson, "On the relative complexities of some geometric problems." in *Proceedings of the 7th Canadian Conference on Computational Geometry (CCCG)*, 1995, pp. 85–90.

[63] ——, "New lower bounds for hopcroft's problem," *Discrete and Computational Geometry*, vol. 16, pp. 389–418, 1996.

[64] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985.

[65] P. M. Camerini, "The min-max spanning tree problem and some extensions," *Information Processing Letters*, vol. 7, pp. 10–14, 1978.

[66] M. J. Atallah, "Dynamic computational geometry," in *Proceedings of 24th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1983, pp. 92–99.

[67] J. Basch, L. J. Guibas, and J. Hershberger, "Data structures for mobile data," *Journal of Algorithms*, vol. 31, pp. 1–28, 1999.

[68] Z. Rahmati and A. Zarei, "Kinetic Euclidean minimum spanning tree in the plane," *Journal of Discrete Algorithms*, vol. 16, pp. 2–11, 2012.

[69] P. K. Agarwal and J. Matoušek, "Dynamic half-space range reporting and its applications," *Algorithmica*, vol. 13, no. 4, pp. 325–345, 1995.

[70] T. M. Chan, "A dynamic data structure for 3-D convex hulls and 2-D nearest neighbor queries," *Journal of the ACM*, vol. 57, pp. 16:1–16:15, 2010.

[71] D. Eppstein, "Dynamic Euclidean minimum spanning trees and extrema of binary functions," *Discrete and Computational Geometry*, vol. 13, pp. 111–122, 1995.

[72] T. M. Chan, "Dynamic geometric data structures via shallow cuttings," *Discrete and Computational Geometry*, vol. 64, pp. 1235–1252, 2020.

[73] G. S. Brodal and R. Jacob, "Dynamic planar convex hull," in *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002, pp. 617–626.

[74] T. M. Chan, "Dynamic planar convex hull operations in near-logarithmaic amortized time," *Journal of the ACM*, vol. 48, pp. 1–12, 2001.

[75] J. Hershberger and S. Suri, "Applications of a semi-dynamic convex hull algorithm," *BIT Numerical Mathematics*, vol. 32, no. 2, pp. 249–267, 1992.

[76] M. H. Overmars and J. van Leeuwen, "Maintenance of configurations in the plane," *Journal of Computer System Sciences*, vol. 23, no. 2, pp. 166–204, 1981.

[77] T. M. Chan, "Optimal partition trees," *Discrete and Computational Geometry*, vol. 47, pp. 661–690, 2012.

[78] J. Matoušek, "Efficient partition trees," *Discrete and Computational Geometry*, vol. 8, pp. 315–334, 1992.

[79] ——, "Range searching with efficient hierarchical cuttings," *Discrete and Computational Geometry*, vol. 10, pp. 157–182, 1993.

[80] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry – Algorithms and Applications*, 3rd ed. Berlin: Springer-Verlag, 2008.

[81] E. A. Ramos, "On range reporting, ray shooting and k-level construction," in *Proceedings of the 15th Annual Symposium on Computational Geometry (SoCG)*, 1999, pp. 390–399.

[82] B. Chazelle, L. J. Guibas, and D. Lee, "The power of geometric duality," *BIT*, vol. 25, pp. 76–90, 1985.

[83] P. K. Agarwal, J. Matoušek, and M. Sharir, "On range searching with semialgebraic sets. ii," *SIAM Journal on Computing*, vol. 42, no. 6, pp. 2039–2062, 2013.

[84] J. Matoušek and Z. Patáková, "Multilevel polynomial partitions and simplified range searching," *Discrete and Computational Geometry*, vol. 54, no. 1, pp. 22–41, 2015.

[85] P. K. Agarwal, Range searching, in *Handbook of Discrete and Computational Geometry,* C.D. Tóth, J. O'Rourke, and J.E. Goodman (eds.), 3rd ed. CRC Press, 2017, pp. 1057–1092.

[86] ——, Simplex range searching and its variants: a review. In *A Journey Through Discrete Mathematics*. Springer, 2017, pp. 1–30.

[87] J. Matoušek, "Geometric range searching," *ACM Computing Survey*, vol. 26, pp. 421–461, 1994.

[88] P. Afshani and P. Cheng, "Lower bounds for semialgebraic range searching and stabbing problems," in *Proceedings of the 37th International Symposium on Computational Geometry (SoCG)*, 2021, pp. 8:1–8:15.

[89] ——, "On semialgebraic range reporting," in *Proceedings of the 38th International Symposium on Computational Geometry (SoCG)*, 2022, pp. 3:1–3:14.

[90] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel, "On the shape of a set of points in the plane," *IEEE Transactions on Information Theory*, vol. 29, no. 4, pp. 551–559, 1983.

[91] B. Chazelle, "On the convex layers of a planar set," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 509–517, 1985.

[92] A. Dumitrescu, A. Ghosh, and C. D. Tóth, "Sparse hop spanners for unit disk graphs," *Computational Geometry: Theory and Applications*, vol. 100, pp. 101 808: 1–14, 2022.

[93] B. Chazelle, "New techniques for computing order statistics in Euclidean space," in *Proceedings of the 1st Annual Symposium on Computational Geometry (SoCG)*, 1985, pp. 125–134.

[94] A. C.-C. Yao, "On constructing minimum spanning trees in k-dimensional spaces and related problems," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736, 1982.

[95] P. K. Agarwal, B. Aronov, M. Sharir, and S. Suri, "Selecting distances in the plane," *Algorithmica*, vol. 9, no. 5, pp. 495–514, 1993.

[96] M. T. Goodrich, "Geometric partitioning made easier, even in parallel," in *Proceedings of the 9th Annual Symposium on Computational Geometry (SoCG)*, 1993, pp. 73–82.

[97] J. Matoušek, "Randomized optimal algorithm for slope selection," *Information Processing Letters*, vol. 39, pp. 183–187, 1991.

[98] B. Chazelle, "Cutting hyperplanes for divide-and-conquer," *Discrete and Computational Geometry*, vol. 9, no. 2, pp. 145–158, 1993.

[99] A. Lubotzky, R. Phillips, and P. Sarnak, "Explicit expanders and the Ramanujan conjectures," in *Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC)*, 1986, pp. 240–246.

CURRICULUM VITAE

Yiming Zhao

Education

- Ph.D. Computer Science. Utah State University, Logan, Utah, USA. August 2019 - April 2023.
- B.E. Software Engineering. South China University of Technology, Guangzhou, Guangdong, China. September 2015 - June 2019.

Research Interests

Computational Geometry, Algorithms and Data Structures, Combinatorial Optimization, Theoretical Computer Science, etc.

Published Journal Articles

- Haitao Wang and Yiming Zhao. "Computing the Minimum Bottleneck Moving Spanning Tree", submitted to Algorithmica, **under review**.
- Haitao Wang and Yiming Zhao. "Reverse Shortest Path Problem for Unit-Disk Graphs", Journal of Computational Geometry, Vol. 14(1), pages 14–47, 2023.
- Haitao Wang and Yiming Zhao. "An Optimal Algorithm for L_1 Shortest Paths in Unit-Disk Graphs", Computational Geometry: Theory and Applications, Vol. 110, Article No. 101960, pages 1–9, 2023.
- Haitao Wang and Yiming Zhao. "Algorithms for Diameters of Unicycle Graphs and Diameter-Optimally Augmenting Trees", Theoretical Computer Science, Vol. 890, pages 192–209, 2021.

- Haitao Wang and Yiming Zhao. "A Linear-Time Algorithm for Discrete Radius Optimally Augmenting Paths in a Metric Space", International Journal of Computational Geometry and Applications, Vol. 30, pages 167–182, 2020.

Published Conference Papers

- Haitao Wang and Yiming Zhao. "Improved Algorithms for Distance Selection and Related Problems", **under review**.
- Haitao Wang and Yiming Zhao. "A Simple Algorithm for Unit-Disk Range Reporting", **under review**.
- Haitao Wang and Yiming Zhao. "Computing the Minimum Bottleneck Moving Spanning Tree", Proceedings of the 47th International Symposium on Mathematical Foundations of Computer Science (MFCS), pages 82:1–82:15, 2022.
- Haitao Wang and Yiming Zhao. "Reverse Shortest Path Problem in Weighted Unit-Disk Graphs", Proceedings of the 16th International Conference and Workshops on Algorithms and Computation (WALCOM), pages 135–146, 2022. **(Best Student Paper Award)**
- Haitao Wang and Yiming Zhao. "An Optimal Algorithm for L_1 Shortest Paths in Unit-Disk Graphs", Proceedings of the 33rd Canadian Conference on Computational Geometry (CCCG), pages 211–218, 2021.
- Haitao Wang and Yiming Zhao. "Reverse Shortest Path Problem for Unit-Disk Graphs", Proceedings of the 17th Algorithms and Data Structures Symposium (WADS), pages 655–668, 2021.
- Haitao Wang and Yiming Zhao. "Algorithms for Diameters of Unicycle Graphs and Diameter-Optimally Augmenting Trees", Proceedings of the 15th International Conference and Workshops on Algorithms and Computation (WALCOM), pages 27–39, 2021.

- Haitao Wang and Yiming Zhao. “A Linear-Time Algorithm for Discrete Radius Optimally Augmenting Paths in a Metric Space”, Proceedings of the 32nd Canadian Conference on Computational Geometry (CCCG), pages 174–180, 2020.