Geometric Search on Point-Tuples

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Akash Agrawal

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILISOPHY

Advisor: Prof. Ravi Janardan

June, 2016

Acknowledgements

"For this relief, much thanks ..."

– Shakespeare, Hamlet.

My deepest gratitude goes to my advisor Professor Ravi Janardan whose insight and incredible knowledge of the field came to my rescue on many occasions. This dissertation would not have been possible without his dedication and guidance; he spent so much time with me, discussing problems and reading and commenting on the manuscripts. He has been a great mentor to me.

I would also like to thank Professor Volkan Isler, Mohamed Mokbel, and Vic Reiner for serving on my thesis committee and providing critical comments, valuable feedbacks, and advise.

My sincere thanks are also due to my friends and fellow graduate students for making these past years a great experience. In the algorithms group, I am grateful to Rahul Saladi, Yuan Li, and Jie Xue for their friendship, support, and countless hours of brainstorming sessions. Outside the group, I am especially thankful to my friends Anuj Karpatne, Ankush Khandelwal, Naveen Elangovan, and Gowtham Atluri for helping me unwind after long days of research.

Last, but not the least, I would like to thank the Department of Computer Science at the University of Minnesota for the generous funding and assisting me throughout these years.

Dedication

To my parents, Sudha and Satish Agrawal.

Abstract

Computational geometry is a branch of computer science that deals with the algorithmic aspects of geometric problems. Nearest neighbor search and range search are two fundamental geometric search problems in computational geometry. Despite the long history of these problems, the majority of the research until now has been focused on simple objects such as points. However, many applications such as trip planning under budget constraint, routing, task allocation, wireless sensor networks, etc. require querying more complex objects, such as point-tuples, that are created at the query time. Motivated by such considerations, this dissertation is devoted to the design of efficient algorithms and data structures for a relatively new class of geometric search problems where the goal is to report point-tuples that satisfy certain query constraints.

Given several disjoint sets of points with an ordering of the sets, two fundamental search problems are addressed. The first problem is a point-tuple version of the nearest neighbor search problem, where the goal is to report the ordered sequence (i.e., tuple) of points (one per set and consistent with the given ordering of the sets) such that the total distance traveled starting from a query point q and visiting the points of the tuple in the specified order is minimum. The second problem is a point-tuple version of the range search problem, where, for a distance constraint δ, the goal is to report all the tuples of points (again, one per set and consistent with the given ordering of the sets) such that, for each tuple, the total distance traveled starting from q and visiting the points of the tuple is no more than δ.

In many applications, the points also have feature attributes in addition to spatial attributes (coordinates). For such datasets, three extensions of the point-tuple version of the range search problem are also studied. These extensions differ in the type of constraint applied on the output tuples, as dictated by the underlying application. The first extension applies range constraints on the feature attribute values. The second extension applies a threshold constraint on the scores of the tuples, computed from the feature attribute values using an aggregation function (e.g., sum, product, etc.). Finally, the third extension seeks the so-called skyline of the output tuples based on the feature attribute values.

Contents

List of Tables

List of Figures

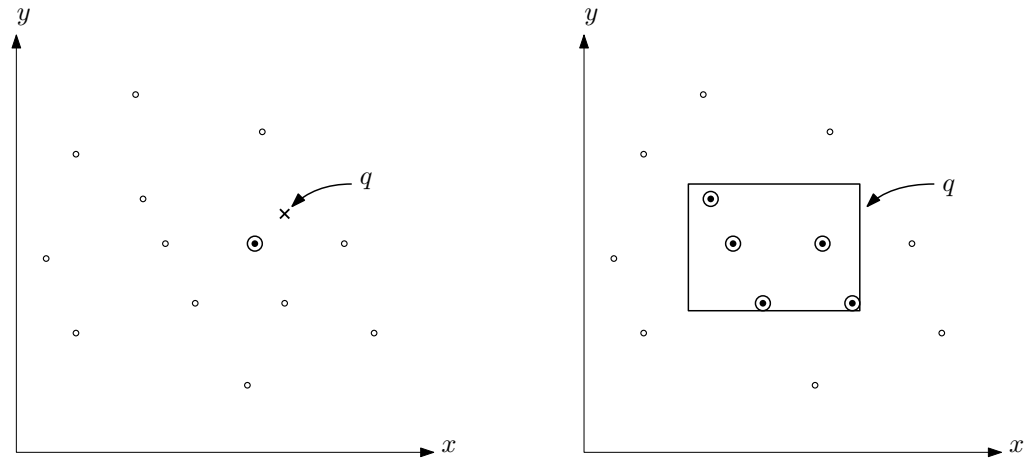
Chapter 1

Introduction

1.1 Motivation

Computational geometry is the branch of computer science which studies algorithms for problems where the data can be modeled as geometric objects (e.g. points, lines, polygons, polyhedra, etc.) and the solution can be formulated in terms of geometric operations on these objects. It has been an active field of research since the mid-1970s (the Ph.D. thesis of Shamos [1] is generally regarded as the genesis of the field) and has applications in a wide range of domains including, but not limited to, spatial databases, computer graphics, computer-aided design and manufacturing, robotics, and geographic information systems [2, 3].

Nearest neighbor search and range search are two fundamental geometric search problems in computational geometry. For a given set S of points in a d-dimensional space, the goal of the nearest neighbor search problem is to report the point of S which is closest (under a suitable distance metric) to any query point (see Figure 1.1a). The goal of the range search problem is to report the points of S lying inside any query object such as a rectangle, ball, halfspace, etc. (see Figure 1.1b). Due to their many applications in diverse domains such as spatial databases, robotics, VLSI design, computer vision, computer graphics, facility location, information retrieval, data mining, etc., the nearest neighbor search problem and the range search problem have been studied extensively in the computational geometry and database literature, and space- and query time-efficient solutions have been devised for many instances of the problems. (See, for



(a) Nearest neighbor query. (b) Range query.

Figure 1.1: Example of the nearest neighbor search query and the range search query problems in the plane. Output points are circled.

instance [2], [3], [4] [5], [6], [7], [8], [9].)

Despite the long history of the nearest neighbor search and the range search problems, the majority of the work so far has been restricted to simple geometric objects such as points. However, recent technological advancements and increase in the use of mobile and web applications have spurred many new problems and challenges that involve querying more complex objects, such as, for instance, *point-tuples*, that are created as a combination of simple objects (points) at query time. For example, consider the following trip planning application.

Suppose that we are given a database that contains locations of venues of different types (e.g., restaurants and theaters) in a large city. A tourist is interested in visiting a restaurant and a theater, in that order, such that the total travel distance is minimized (due to reasons such as the cost of travel, time, mileage restriction etc.). She wishes to identify a (restaurant, theater) tuple, among the set of all such tuples, so that, the distance from her current location to the restaurant plus the distance from the restaurant to the theater is smallest. Therefore, she issues such a query from her location using, say, an application running on her smartphone and gets back the (restaurant, theater) tuples satisfying the given constraint. This example generalizes naturally to more than two types of venues (restaurants, theaters, gas stations, grocery stores, etc.), where

the objective is to start from the query point and visit one venue of each type, in a pre-specified order, so that the total travel distance is minimized.

By modeling the venues of different types as points of different colors in the plane, the problem of finding the facility tuple with minimum travel distance can be modeled as the following geometric problem: We are given a set S of n points in the plane, where each point is assigned a color from a given palette of m colors and there is an ordering of the colors. The goal is to preprocess S so that for any query point q, we can report the ordered sequence (i.e., tuple) of points (one per color and consistent with the given ordering of the colors) such that the distance traveled from q to visit the points of the tuple is minimized.

The above-mentioned geometric problem is a point-tuple version of the nearest neighbor search problem with the travel distance used as the distance metric. We call this problem the *nearest neighbor color tuple query* (NNCTQ) problem. Similarly, point-tuple versions of other geometric search problems can be defined naturally for various applications such as routing, task allocation, wireless sensor networks, etc. by using suitable constraints. For example, a point-tuple version of the range search problem can be defined by using a range constraint on the travel distance.

Motivated by such applications, in this dissertation, we study a class of geometric search problems where the goal is to report point-tuples that satisfy query constraints. We refer to this (mostly) new class of problems as *geometric search on point-tuples*. The rest of the chapter is organized as follows. We describe the problems studied in this dissertation in Section 1.2. In Section 1.3, we present a brief survey of the related works. We conclude this chapter with the organization of the remainder of the dissertation in Section 1.4.

1.2 Results in this dissertation

In this dissertation, we study the point-tuple versions of the nearest neighbor search and the range search problems. The problems studied in this dissertation can be broadly grouped into three categories: nearest neighbor search, range search, and extensions to handle additional constraints. Throughout, we consider a given set S of n points in the plane, where each point is assigned a color from a given palette of m colors and there

is an ordering of the colors. Also, we consider m-tuples $t = (p_1, \ldots, p_m) \in S^m$ with the property that p_i has color c_i, $1 \leq i \leq m$, so, henceforth we will not state this explicitly.

1.2.1 The nearest neighbor search problem

Here, we study the NNCTQ problem mentioned in Section 1.1 which is the point-tuple version of the nearest neighbor search problem. It is worth mentioning that the naïve approach of solving this problem requires exploring all tuples. For example, consider Figure 1.2 which depicts a small data-set containing points of two colors, blue points b_i and red points r_j. The query point is denoted as q.

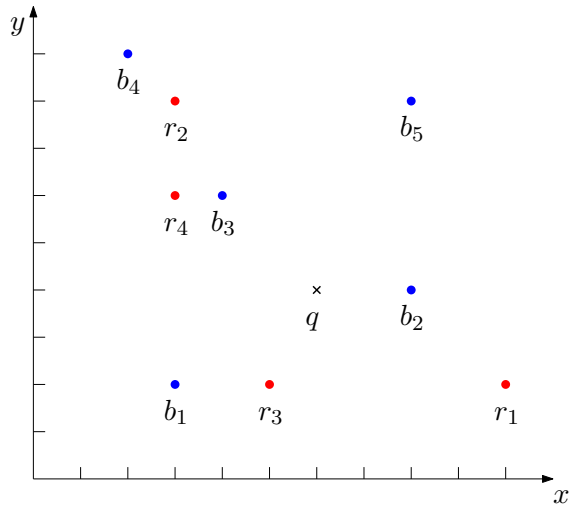


Figure 1.2: An example of data-set containing points of two colors (red and blue points). Point 'q' is the query point. (All figures are best viewed in color.)

The optimal (blue, red) tuple, here, is (b_3, r_4) for which the total travel distance from q is 3.8 units. By contrast, the total distance for the tuple (b_2, r_3), where b_2 and r_3 are the closest blue and red points to q, respectively, is 5.6 units. It is also worth noting that a greedy approach that picks the nearest blue point to q and nearest red point to that blue point yields (b_2, r_1), which has total distance 4.8 units.

We present several efficient algorithms for this problem that are of both practical and theoretical interest. Our key contributions for this problem are as follows. First, we show an approach to transform (in preprocessing) the set of colored points to a set of weighted points in such a way that the answer to a certain nearest neighbor query on

weighted points, where the underlying distance function is a combination of the distance and the weight, yields the answer to the NNCTQ problem. We call the latter problem a *weighted nearest neighbor search query* problem. Second, we present three algorithms to solve the weighted nearest neighbor search query problem.

1.2.2 The range search problem

Here, we consider a point-tuple version of the circular range search problem. For any query point q and a distance budget δ, the goal of this problem is to report all point-tuples of the given ordering of the colors such that the distance traveled from q to visit the points of each tuple is at most δ. We call this the *colored tuple range query* (CTRQ) problem.

A naïve solution for this problem is to first find all the points that are within δ distance from the query point and then check this set for the tuples to output. Unfortunately, this can be very expensive. For example, consider Figure 1.3 which depicts a small data-set containing points of two colors, blue points b_i and red points r_j, in the plane. The query point is denoted by q and the distance constraint is represented as a disk of radius δ centered at q. Let us assume that $\delta = 4$ units.

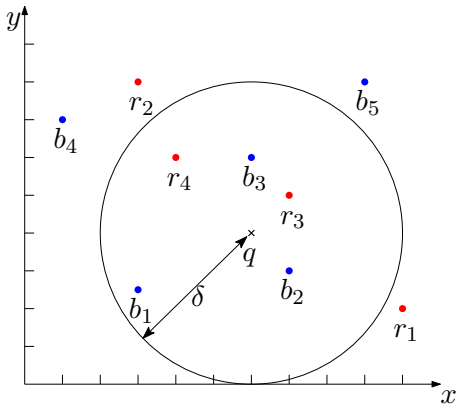


Figure 1.3: An example of data-set containing points of two colors (red and blue points). Point 'q' is the query point. The disk with radius δ represents the maximum travel distance constraint.

In this example, it is clear that even though points b_1, b_2, b_3, r_3, and r_4 are all reachable within travel distance δ, only the (blue, red) tuples $(b_2, r_3), (b_3, r_3)$, and (b_3, r_4)

with total travel distance $3.4, 3.4$, and 4, respectively, are part of the answer set. The other (blue, red) tuples formed by the points reachable from q within travel distance δ have total travel distance more than δ. (The other possible (blue, red) tuples are $(b_2, r_4), (b_1, r_4)$, and (b_1, r_3) with total travel distance $5.7, 7$ and 8.1, respectively.)

Now consider a scenario where all the blue points lie on the perimeter of a disk centered at q and of radius δ and all the red points lie inside the disk. It is clear that all the points are reachable from q and yet the total travel distance for every (blue, red) tuple is larger than δ, so that the answer set is empty! This is the worst case scenario for the naïve solution.

Depending on the application there are two versions of the CTRQ problem that are of interest. In the first version, δ is known beforehand (during preprocessing) while in the second version δ is known only at query time. (In both versions, q is known only at query time.) We will refer to these versions as the *fixed distance* CTRQ and the *variable distance* CTRQ problems, respectively.

For example, in a trip planning application, some car rental companies place a restriction on the maximum distance a user can travel. In this case the maximum travel distance, δ, is fixed for all the users and is pre-specified. This is an instance of the fixed distance problem. On the other hand, for some applications, the distance constraint can be imposed by reasons like travel time, travel cost etc. In this scenario, the maximum travel distance, δ, depends on the user and is specified only during the query. This is an instance of the variable distance problem. One might suspect that the fixed distance problem admits a more efficient solution than the variable distance problem and, as we will see, this is indeed the case.

The challenge in this problem is to be able to identify *precisely* those tuples that need to be reported—no more, no less. Towards this end, we have identified a novel two-step approach. In the first step, we identify a subset of points of the first color that are guaranteed to be part of at least one reported tuple; we call these points *fruitful points*. In the second step, we recursively explore the points of the remaining colors, starting from each fruitful point, to identify the desired tuples.

There are two key ideas underlying our solution. First, we are able to process the data in such a way that the time required to find the fruitful points is linear in their number, while only using near-linear storage. Second, we are able to tune the query

algorithm to the (unknown) number of output tuples to further reduce the query time.

1.2.3 Handling additional feature attributes

The tuple identification in the CTRQ problem mentioned above is based on spatial attributes or coordinates of the points in the plane. However, in many applications, the data points contain additional feature attributes, and applying constraints on these feature attributes in addition to the distance constraint can provide more meaningful results. For instance consider the trip planning example mentioned in the Section 1.1. In a more realistic scenario of this example, each restaurant has a rating and an average cost and each theater has showtimes and ratings for the shows, and the tourist, in addition to the distance constraint, may prefer to visit only those restaurants that have 3 star$-$4 star rating with average cost \$10$-$\$25, and only those theaters that have showtimes between 7:00 p.m.$-$9:00 p.m and rating for the show more than 3.5 stars. As another example, the tourist may wish to identify the tuples such that each tuple satisfies the distance constraint and the average ratings of the facilities in the tuple is more than a certain threshold value (e.g., 3.5 stars). Finally, the user may want to prune a potentially large answer set by retaining only those tuples that are clearly better than other tuples in terms of their feature attributes.

Motivated by the above considerations, for points that have feature attributes in addition to spatial coordinates, we study three extensions of the CTRQ problem. Specifically, given set S of n points in the plane, where each point has associated additional feature attribute values and is assigned a color from a palette of m colors. Also, given is an ordering of the colors. First, we study an extension of the CTRQ problem where the user can specify additional range constraints, at query time, on the feature attribute values of the points of different colors. The next extension we study is that of applying a query threshold constraint on the scores of the tuples that satisfy the distance constraint. Here, the score of a tuple is computed as an aggregation (e.g., sum, product, etc.) of the feature attribute values of the points in the tuple. Finally, we extend the CTRQ problem to the problem of reporting the so-called *skyline* (or, *Pareto-optimal* set) of the tuples that satisfy the distance constraint. Here, the skyline is computed based on the feature attribute values of the points in the tuples. (The skyline is defined formally in Section 1.2.3.3.)

1.2.3.1 CTRQ problem with feature constraints

Here we extend the CTRQ problem to apply range constraints on feature attribute values in addition to the distance constraints during tuple identification. Specifically, the points have feature attribute values (in addition to spatial coordinates) and the problem we wish to solve is as follows. For any query point q, a distance budget δ, and a set of range constraints $\mathcal{Z}$ on the feature attribute values of the points of different colors, report all tuples such that the distance traveled from q to visit the points of each tuple is at most δ and the points in the tuples satisfy the corresponding range constraints on the feature attributes. We call this the *feature constraint color tuple range query* (ConCTRQ) problem.

Similar to the CTRQ problem, there are two versions of the ConCTRQ problem that are of interest; *fixed distance* and *variable distance* ConCTRQ problems, where δ is known beforehand (during preprocessing) or is specified at the query time, respectively. (The query point q and the set of feature constraints $\mathcal{Z}$ are known only at query time.)

A naïve solution to this problem is to first find all the points that satisfies the feature constraints (using a range search [2, 8]) and then check this set for the tuples to output. Unfortunately, this approach is not efficient. For example, consider Figure 1.4 which depicts a small data-set containing points of two colors. The location of the points are shown in Figure 1.4a. Here, q denotes the query point and the disk of radius δ centered at q represents the maximum travel distance constraint. The feature attribute values of blue and red points are shown in Figure 1.4b and 1.4c, respectively. Here, the rectangle Z_B (resp., Z_R) represents the constraints on the feature attribute values of the blue (resp., red) points. Let us assume that $\delta = 4$ units.

In this example, it is clear that even though points b_3, b_5, r_3, and r_5 satisfy the constraints on feature attribute values, only the (blue, red) tuple (b_3, r_3) with total travel distance 3.4 is part of the answer set. The other (blue, red) tuples formed by the points that satisfy the constraints on feature values ($(b_3, r_5), (b_5, r_3)$, and (b_5, r_5)) have total travel distance more than δ.

Another naïve solution for the problem is to first find all tuples that satisfy the distance constraint (using the solution for the CTRQ problem in Chapter 3 (see also [10])), and then check the points in each tuple against the constraints on feature values. Again, this approach in not efficient. In the above-mentioned example (Figure 1.4), it is clear

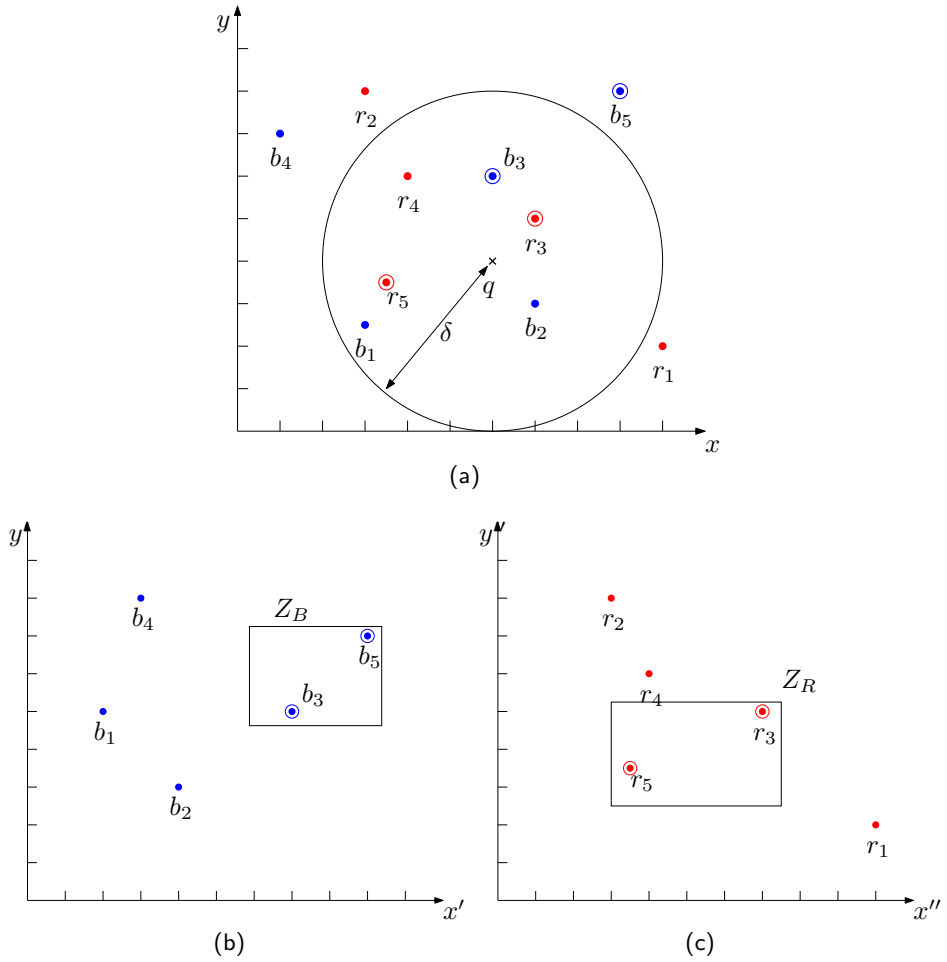


Figure 1.4: An example of ConCTRQ problem instance. (a) Location of the points of two colors. Point q is the query point. The disk with radius δ represents the maximum travel distance constraint. Circled points represent the points that satisfy the constraints on feature attribute values. (b) Feature attribute values of the blue points. The rectangle Z_B represents the feature constraint for the blue points. (c) Feature attribute values of the red points. The rectangle Z_R represents the feature constraint for the red points.

that even though the (blue, red) tuples $(b_2, r_3), (b_3, r_3)$, and (b_3, r_4) with total travel distance 3.4, 3.4, and 4, respectively, are part of the answer set for CTRQ, only the (blue, red) tuple (b_3, r_3) contains points that satisfy the constraints on feature values. Now consider the scenario depicted in Figure 1.5. In this scenario, both of the naïve approaches will explore a large number of tuples even though the answer set is empty! This is the worst case scenario for the naïve approaches.

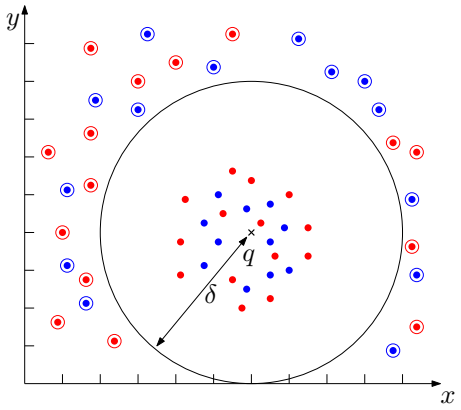


Figure 1.5: Worst case example for the naïve approaches. Point q is the query point. The disk with radius δ represents the maximum travel distance constraint. Circled points represent the points that satisfy the constraints on feature value.

Our first key contribution for this problem is a proof that a certain minimum amount of storage is unavoidable to achieve a fast query time. Our second key contribution is an efficient algorithm that uses no more than this amount of storage.

1.2.3.2 CTRQ problem with threshold constraints

In this work, we extend the concept of the CTRQ problem to apply a threshold constraints (τ) on the scores of the tuples that satisfy the distance constraint (δ). Here, the score of a tuple is computed by applying some aggregation function (such as sum, product, etc.) on the feature attribute values of the points in the tuple. We call this the *threshold constraint color tuple range query* (τ-CTRQ) problem.

Depending upon the application, $\delta > 0$ and $\tau \geq 0$ can both be "fixed" (known during preprocessing) or "variable" (specified only at query time), resulting in four

versions of the problem that are of interest. We call these *fixed-distance fixed-threshold*, *fixed-distance variable-threshold*, *variable-distance fixed-threshold*, and *variable-distance variable-threshold* τ-CTRQ problems.

A naïve solution for this problem is to first find all the tuples that satisfy the distance constraint (using the solution for the CTRQ problem), and then check the score of each tuple against the threshold constraint. This approach is not output-sensitive and hence, is quite inefficient. For example, consider the data-set containing two-colored points shown in Figure 1.6. Each point is associated with a single attribute denoting the score of the point. Again, q denotes the query point and the disk of radius δ centered at q represents the maximum travel distance constraint. Let the score of a tuple be the sum of scores of the points in the tuple.

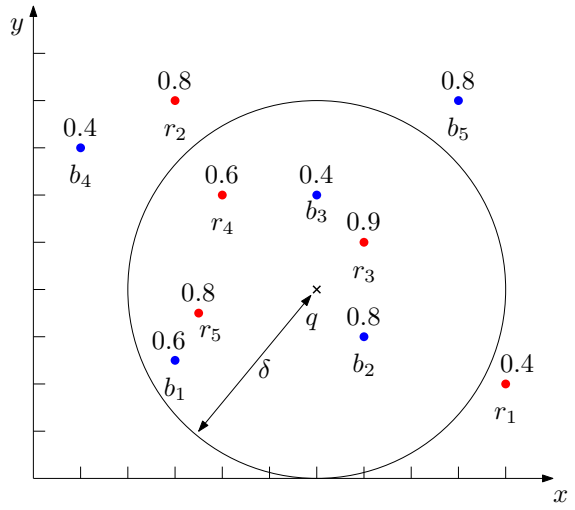


Figure 1.6: An example data-set containing points of two colors (red and blue points). Each point is associated with a single attribute denoting the score of the point. Point 'q' is the query point and the disk with radius δ represents the maximum travel distance constraint.

For $\delta = 4$ units and $\tau = 1.2$, it is clear that out of the three (blue, red) tuples that are part of the answer set for CTRQ ((b_2, r_3), (b_3, r_3), and (b_3, r_4) with total travel distances $3.4, 3.4$, and 4, respectively), only the (blue, red) tuples (b_2, r_3) and (b_3, r_3) with score 1.7 and 1.3, respectively, satisfy the threshold constraint. Now consider a scenario where a large number of tuples satisfy the distance constraint but the threshold constraint is larger than the highest score of any tuple. In this scenario, the naïve approach will

explore a large number of tuples even though the answer set is empty! This is the worst case scenario for the naïve approach.

For this problem, similar to the CTRQ problem, we have identified a two-step approach. In the first step, we identify the set of fruitful points, i.e. a subset of points of the first color that are guaranteed to be part of at least one reported tuple. In the second step, we recursively explore the points of the remaining colors, starting from each fruitful point, to identify the desired tuples.

1.2.3.3 Reporting the tuples in the skyline of tuples

In d-dimensional coordinate space a point p *dominates* a point p' if each coordinate of p is at least as large as the corresponding coordinate of p', with inequality in at least one dimension. The *skyline* of a given set of points is a subset of points where no point is dominated by any other point of the given set. The skyline (also known as *Pareto set* or *maximal vector*) is useful in multi-criteria decision-making on large data-sets as it yields a set of viable candidates for further exploration. It has been well-studied in the database, optimization, and computational geometry literature among others (see, for instance [3, 11, 12, 13, 14, 15, 16, 17, 18, 19]).

In this work, we consider colored point-sets where each point has a single real-valued feature attribute value (in addition to its spatial coordinates). For such data-sets, we investigate a version of the CTRQ problem where the goal is to report the tuples that satisfy the distance constraint (δ) and, moreover, are not dominated by any other tuples that satisfy δ. Here a tuple t is *dominated* by another tuple t' if and only if the feature attribute value of each point in t is less than or equal to the feature attribute value of the point of the same color in t', with strict inequality for at least one point. We call this the *colored tuple range skyline query* (SkyCTRQ) problem. The SkyCTRQ problem is essentially a point-tuple version of the *range-skyline query* problem [20, 21]. Again, we study the *fixed distance* and the *variable distance* versions of the problem, where δ is known beforehand (during preprocessing) or is known only at the query time, respectively.

A naïve solution for the SkyCTRQ problem is to first find all the tuples that satisfy the distance constraint (using the solution for CTRQ problem), and then compute the skyline of the reported tuples. This can be very expensive. For example, in the data-set

of Figure 1.6, even though the (blue, red) tuples $(b_2, r_3), (b_3, r_3)$, and (b_3, r_4) with total travel distance 3.4, 3.4, and 4, respectively, are part of the answer set for CTRQ, only the (blue, red) tuple (b_2, r_3) is in the answer set for the SkyCTRQ problem.

Our key contributions for this problem are as follows. We first introduce a more general problem, where the goal is to report the skyline of the points that lie in a query halfspace. We call this the Halfspace-Range-Skyline query problem. We present several efficient solutions for this problem. We then introduce a geometric transformation that efficiently maps the SkyCTRQ problem to the Halfspace-Range-Skyline query problem, thereby yielding efficient solution for the SkyCTRQ problem.

1.3 Sampling of related work

1.3.1 Nearest neighbor search problem

As mentioned earlier, the nearest neighbor query problem has been well studied and several theoretical and practical solutions have been proposed in both the exact and the approximate settings. In the exact setting, assuming that data points are indexed using a spatial access method (such as an R-tree or kd-tree [2, 22]), several pruning techniques have been proposed to restrict the search space [22, 23, 24, 25]. These algorithms can be easily extended to report $k > 1$ nearest neighbors [22, 26]. Unfortunately, the performance of these algorithms is highly dependent on the distribution of the data points and the location of the query point [22]. For points in $\mathbb{R}^2$, a Voronoi Diagram can be used to perform nearest neighbor search [27]. For n data points, the Voronoi Diagram approach requires $O(n \log n)$ preprocessing time, uses $O(n)$ space, and supports nearest neighbor queries in $O(\log n)$ time. The Voronoi Diagram-based solution can also be extended to report $k > 1$ nearest neighbors, for a fixed value of k, by using an order-k Voronoi Diagram [28]. Unfortunately, this approach takes $O(k^2(n-k))$ space, which can be high for some values of k (e.g. $O(n^3)$ for $k \approx n/2$).

The nearest neighbor problem has also been studied in higher dimensions. Since exact solutions tend to be expensive, attention has been focused on approximate solutions. Arya et al. [29] proposed a generalized space-time trade-off for approximate nearest neighbor search; given a trade-off parameter γ, where $2 \leq \gamma \leq 1/\varepsilon$, ε-approximate nearest neighbor queries can be answered in $O(\log(n\gamma) + 1/(\varepsilon\gamma)^{(d-1)/2})$ time using an

index structure of size $O(n\gamma^{d-1}\log(1/\varepsilon))$.

Other versions of the nearest neighbor query have also been studied in the GIS and spatial database literature. For example, Papadias et al. [26] presented the aggregate nearest neighbor query where the input consists of a set of data-points and for a given set of query points, the goal is report a point from the data-set that minimizes the aggregate distance (min, max, or sum) from the query points. Aly et al. [30] presented algorithms to efficiently evaluate spatial queries that uses two kNN predicates. The nearest neighbor problem has also been studied for other distance metrics, e.g. network distance metric, where the data points are specified over vertices of a network and the distance between two points is defined as the length of shortest path connecting the corresponding vertices [31, 32, 33].

1.3.2 Range search problem

The range search problem is a fundamental query retrieval problem. Due to its wide range of application in diverse domains, the range search query problem has been extensively studied in computational geometry and database literature and several efficient solutions have been proposed for different instances of this problem (see, for instance, [2, 3, 7, 8, 9]). The CTRQ problem is closely linked to the circular range query problem, whose goal is to preprocess a given set of n points in $\mathbb{R}^d$ so that the points inside a query ball can be reported efficiently. Due to applications in proximity queries, the circular range query problem has been studied extensively [3]. Furthermore, via a standard lifting transformation [34], the circular range query problem in $\mathbb{R}^d$ can be transformed into a halfspace range reporting problem in $\mathbb{R}^{d+1}$. In [35], Afshani and Chan have shown that the halfspace range reporting problem in $\mathbb{R}^d$, $d \geq 4$, can be solved in $O(n^{1-1/\lfloor d/2 \rfloor}\log^{\mathrm{O}(1)} n + k)$ query time using an $O(n)$-space data structure, where k is the output size, which is the most efficient solution known so far. Moreover, as shown in [35], the halfspace range search problem in $\mathbb{R}^3$ (hence circular range search in $\mathbb{R}^2$) can be solved in $O(n)$ space and $O(\log n + k)$ query time.

1.3.3 Range skyline query problems

For a given set of points O, the goal of a range skyline query problem (also known as a *range maxima query* problem) is to report the skyline of the points of O lying inside any query object such as rectangle, halfspace, etc. For the points in the plane, early research on this problem focused on dominance queries (skyline of points that dominate a query point) and axis-parallel query lines (skyline of points on specified side of the axis-parallel query line). It has been shown that both of these problems can be solved optimally in $O(\log n + k)$ time using a $O(n)$ space data structure, where k is the output size [36, 37, 38, 39, 40]. For three sided query rectangles in the plane, Brodal et al. [41] presented an optimal dynamic data structure that takes $O(n)$ space and $O(\log n)$ time per update and answers the top-open queries in $O(\log n + k)$ time. For the general query rectangles (i.e., 4-sided rectangles in the plane), their solution takes $O(n \log n)$ space with $O(\log^2 n)$ update time and $O(\log^2 n + k)$ query time. Kalavagattu et al. [42] showed that if the insertion and deletion of points are not allowed then the range skyline queries for the 4-sided query rectangles can be answered in $O(\log n + k)$ time using $O(n \log n)$.

The range skyline query problem in the plane has been studied for other computational models as well. In the word RAM model, Brodal et al. [41] presented a solution for 3-sided query rectangles that takes $O(n)$ space, $O(\frac{\log n}{\log \log n} + k)$ query time, and $O(\frac{\log n}{\log \log n})$ update time. For the static case, Das et al. [43] showed that the problem can be solved for 4-sided query rectangles in $O(\frac{\log n}{\log \log n} + k)$ query time using $O(n \frac{\log n}{\log \log n})$ space word RAM structure. In the external memory model, Kejlberg-Rasmussen et al. [20] showed that the any solution for the problem with 4-sided query requires $\Omega(n \log_B n / \log \log_B n)$ space to achieve a query time of $O(\log_B n + k/B)$ I/Os, where B is the block size. (See also, [44, 21] for the other versions of the problem.)

There are two major differences between our work on the range skyline query problem and the current state of the art. First, we study the problem for halfspace query ranges while the existing works have been restricted to orthogonal query ranges. Second, we consider different spaces for the range restriction and the skyline computation, while the existing works compute the skyline on the same space where the range restriction is applied.

1.3.4 Point-tuple search problems in databases

In the context of spatial databases, the NNCTQ problem has been studied as the optimal sequenced route (OSR) query problem [45] (also called multi-type nearest neighbor query [46] or transitive nearest neighbor query [47, 48]). For this problem, Sharifzadeh et al. [45] proposed three algorithms, LORD, R-LORD, and PNE that utilize certain threshold values to filter out the non-candidate points. The proposed PNE technique can be adapted to report top-k optimal routes also.

A version of the OSR problem where the order of visit to the sets is not specified is proven to be NP-hard and is known as the trip planning query (TPQ). In [49], Li et al. have developed approximation algorithms for the instance of TPQ where a source and a destination are specified. Kanza et al. [50] have shown polynomial-time heuristics for the constrained version of the TPQ problem. In [51], Chen et al. have proposed a generalization of the OSR query, called multi-rule partial sequenced route (MRPSR) query, where the order of traversal is defined by a partial order. Due to its practical applications, the OSR problem has been studied extensively and practical solutions have been proposed for many different versions of the problem such as OSR on road networks [52], k-OSR [53], interactive route search [54], traffic-aware OSR [55], and many others [56, 57, 58, 59, 60].

All of the above methods are empirical in nature. On the other hand, the NNCTQ method proposed here is based on a rigorous theoretical foundation and offers both exact and guaranteed-quality approximate solutions.

1.4 Organization

The remainder of this dissertation is organized as follows. We formalize and present the solutions for the NNCTQ problem in Chapter 2 and for the CTRQ problem in Chapter 3. The three extensions of the CTRQ problem, namely the ConCTRQ, the τ-CTRQ, and the SkyCTRQ problems are discussed in Chapter 4, Chapter 5, and Chapter 6, respectively. We conclude in Chapter 7 with a brief review of the contributions of this dissertation and discussion of future work.

Chapter 2

Nearest Neighbor Colored Tuple Query (NNCTQ)

Nearest neighbor search is a fundamental geometric query problem, where the goal is to preprocess a set of points so that the one that is closest to a query point can be reported efficiently [2, 4, 6, 7, 33]. In this chapter we study the point-tuple version of the nearest neighbor search problem (Section 1.2.1). We present several efficient algorithms for this problem by first transforming it to a weighted nearest neighbor search problem and then solving the latter using different methods.

2.1 Problem formulation and contributions

Problem 2.1 (NNCTQ)**.** *Let S be a set of n points in $\mathbb{R}^2$, where each point is assigned a color c_i from a palette of m colors ($m \geq 2$). Let C_i be the set of points of color c_i and let $n_i = |C_i|$. Note that $C_i \cap C_j = \emptyset$ if $i \neq j$ and $\bigcup_{i=1}^{m} C_i = S$. Let $CS = (c_1, \ldots, c_m)$ be a given ordering of the colors. We wish to preprocess S into a suitable data structure so that for any query point q in $\mathbb{R}^2$ the m-tuple $P = (p_1, \ldots, p_m)$ can be reported, where $p_i \in C_i$, such that $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i)$ is minimum over all m-tuples in $C_1 \times \cdots \times C_m$. (Here $d(\cdot, \cdot)$ denotes Euclidean distance.)*

In this chapter, we design several algorithms for the NNCTQ problem. In essence, our approach consists of two steps: In the first step, we show how the NNCTQ problem

on S can be transformed (in preprocessing) to the problem of answering a nearest neighbor query for q on the set C_1, where each point of C_1 now has a weight and the underlying distance function is a combination of the distance function $d(\cdot,\cdot)$ and the weight. We call the latter problem a *weighted nearest neighbor query* problem. In the second step, we show how to answer the weighted nearest neighbor query efficiently.

To solve the weighted nearest neighbor query problem, we present three algorithms. The first algorithm, based on a thresholding method, assumes the data points are indexed using a spatial data structure such as a kd-tree or R-tree [2, 7, 22] (which is often the case in many database applications) and uses a pruning technique to restrict the search space. This method, which appears to be quite practical, uses $O(n)$ space, but the query time depends on the configuration of the query and the data points. The second algorithm, based on a geometric transformation, reduces the weighted nearest neighbor query to a standard nearest neighbor query problem in $\mathbb{R}^3$. We show that this method, which uses $O(n)$ space and has a query time of $O(\log n)$, gives a provably good approximation to the optimal solution for NNCTQ. The third algorithm, based on additively-weighted Voronoi Diagrams [61], uses $O(n \log n)$ preprocessing time and $O(n)$ space to build a data structure and can answer the query in $O(\log n)$ time. This result shows what is achievable theoretically. However, it uses heavy-duty machinery (weighted Voronoi Diagrams, point location, persistent search, etc.) and we do not expect it to be competitive in practice with the other methods.

2.2 Preliminaries

Voronoi Diagram and Weighted Voronoi Diagram. The *Voronoi Diagram*, $\mathcal{V}(S)$, of a set, S, of n points in $\mathbb{R}^2$ is a subdivision of the plane into n *cells*, one for each point $p_i \in S$, with the property that a point q lies in the interior of a cell corresponding to point p_i iff $d(q,p_i) < d(q,p_j)$ for all $p_j \in S, p_j \neq p_i$, where $d(\cdot,\cdot)$ is a suitable distance function (e.g., Euclidean distance). The cell corresponding to point p_i is denoted by $\mathcal{V}(p_i)$ and point p_i is called the *generator* of $\mathcal{V}(p_i)$. For any two points $p_i, p_j \in S, p_i \neq p_j$, the locus of all points q such that $d(q,p_i) = d(q,p_j)$ is a straight line that partitions the plane into two closed halfplanes. We denote the halfplane containing p_i by $h(p_i,p_j)$ and the halfplane containing p_j by $h(p_j,p_i)$. $\mathcal{V}(p_i)$ is the intersection of halfplanes $h(p_i,p_j)$,

for all $p_j \in S, p_j \neq p_i$. The Voronoi Diagram for n points can be build in $O(n \log n)$ time and uses $O(n)$ space [27]. (See Chapter 7 in [2].)

If each point $p_i \in S$ has an associated weight $w(p_i)$ then we can define the *Weighted Voronoi Diagram*, $\mathcal{V}_w(S)$, as a subdivision of $\mathbb{R}^2$ into n cells, one corresponding to each point $p_i \in S$ (p_i is the *generator* of the cell) such that for any point q in the interior of p_i's cell, $d(q, p_i) + w(p_i) < d(q, p_j) + w(p_j)$ for all $p_j \in S, p_j \neq p_i$. As shown in [62], the locus of all points q such that $d(q, p_i) + w(p_i) = d(q, p_j) + w(p_j)$ depends on the relative position of p_i and p_j and their corresponding weights, and can be a straight line, a ray, a hyperbolic arc etc. (Details can be found in [62].) The weighted Voronoi Diagram for n points can be constructed in $O(n \log n)$ time and uses $O(n)$ space [61].

Nearest neighbor search. As shown by Shamos *et al.* [27], the point of S closest to a query point q can be found by locating the cell of $\mathcal{V}(S)$ that contains q and reporting the associated generator. This requires constructing an efficient point location data structure for $\mathcal{V}(S)$. Many such data structures are available [63, 64, 65, 66]. For a planar subdivision of size $O(n)$ (e.g. $\mathcal{V}(S)$) these structures can be build in $O(n \log n)$ time using $O(n)$ space and return the cell containing q in $O(\log n)$ time. We propose using the approach taken in [66]. This involves partitioning the plane into vertical strips, by drawing vertical lines through each vertex. This creates a total order of the segments (subdivision edges) within each strip that allows efficient searching. The segments associated with all strips are stored compactly using a persistent red-black tree [66]. To locate q, two binary searches are done—one to determine the strip containing q and the other to locate the (portion of the) cell within the strip that contains q.

A practical approach for reporting the nearest neighbor of a query point q is to use a spatial access method (such as an R-tree [67] or kd-tree [22]) to store the points of S and utilize a pruning method to restrict the search space. The structure can be searched for the nearest neighbor in a depth-first (DF) manner or a best-first (BF) manner [22, 23, 24, 25, 68]. For a given query point, q, a BF search on an kd-tree, similar to the approach proposed by Hjaltason *et al.* [68] for R-tree, uses a min-heap, H, to store the distance from q to the entries of the kd-tree (bounding box or point) visited so far. At each step, the algorithm extracts the kd-tree node in H with minimum distance from q and explores that node. The search starts at the root of the kd-tree and

when it visits a node, it inserts the entry of that node (bounding box or point) into H with its minimum distance to q. When the search reaches the leaf node, the algorithm reports the point contained in the leaf node. The space required for this approach is dominated by the space required for the index structure, which is $O(n)$, and the query time depends on the configuration of the query and data points [22, 69]. This approach can be easily extended to report the $k > 1$ nearest neighbors of q by maintaining an ordered list of the k closest entities seen so far and updating this list as the search progresses [26].

2.3 From colored point-sets to a weighted point-set

As mentioned in Section 2.1, the first step is to preprocess the m given sets of colored points into a single set of suitably-weighted points. We now discuss this in detail. We begin with the following result. (In what follows, we will sometimes refer to points of S as "query" points, even though they are not true query points like q.)

Lemma 2.1. *For some query point q, let $P = (p_1, \ldots, p_m)$ be an m-tuple of $C_1 \times \cdots \times C_m$ that minimizes $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i)$. Then for "query" point p_i, $1 \leq i \leq m$, $P_{i+1} = (p_{i+1}, \ldots, p_m)$ is an $(m-i)$-tuple of $C_{i+1} \times \cdots \times C_m$ that minimizes $d(p_i, p_{i+1}) + \sum_{j=i+1}^{m-1} d(p_j, p_{j+1})$.*

Proof. Assume that P_{i+1} is not the optimal solution for "query" point p_i and let $P'_{i+1} = (p'_{i+1}, \ldots, p'_m) \in C_{i+1} \times \cdots \times C_m$ be optimal. Then $d(q, p_1) + \sum_{j=1}^{i-1} d(p_j, p_{j+1}) + d(p_i, p'_{i+1}) + \sum_{j=i+1}^{m-1} d(p'_j, p'_{j+1}) < d(q, p_1) + \sum_{j=1}^{i-1} d(p_j, p_{j+1})$
$+ d(p_i, p_{i+1}) + \sum_{j=i+1}^{m-1} d(p_j, p_{j+1}) = d(q, p_1) + \sum_{j=2}^{m} d(p_{j-1}, p_j)$, which contradicts the optimality of P. □

Lemma 2.1 implies that for "query" point p_{m-1}, the point p_m minimizes $d(p_{m-1}, p_m)$, i.e., p_m is a nearest neighbor of p_{m-1}. Similarly, for "query" point p_{m-2}, the point p_{m-1} minimizes $d(p_{m-2}, p_{m-1}) + d(p_{m-1}, p_m) = d(p_{m-2}, p_{m-1}) + w(p_{m-1})$, where $w(p_{m-1}) = d(p_{m-1}, p_m)$ is a *weight* assigned to p_{m-1}. In general, for "query" point p_i, $1 \leq i \leq m$, the point p_{i+1} minimizes $d(p_i, p_{i+1}) + w(p_{i+1})$, where $w(p_{i+1}) = \sum_{j=i+1}^{m-1} d(p_j, p_{j+1})$ is the weight of p_{i+1}, i.e., $w(p_{i+1})$ is the length of the shortest path from p_{i+1} that visits one point from each of $C_{i+2}, \ldots, C_m$. (We take $w(p_m) = 0$.) Thus, p_{i+1} minimizes

the sum of $w(p_{i+1})$ and the Euclidean distance between p_i and p_{i+1}. We call p_{i+1} the *Euclidean weighted nearest neighbor* of p_i. Note that $w(p_i) = d(p_i, p_{i+1}) + w(p_{i+1}) \geq 0$.

Thus, for query point q, we have $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i) = d(q, p_1) + w(p_1)$, where $w(p_1) = \sum_{i=2}^{m} d(p_{i-1}, p_i)$ is a weight assigned to p_1. Therefore, the problem of computing the optimal m-tuple $P = (p_1, \ldots, p_m) \in C_1 \times \cdots \times C_m$ for q has been transformed to computing the nearest neighbor $p_1 \in C_1$ for q under the *weighted Euclidean distance function* $d^w(q, p_1) = d(q, p_1) + w(p_1)$, i.e., p_1 is the *Euclidean weighted nearest neighbor* of q.

2.3.1 The preprocessing algorithm

The preprocessing algorithm is based on the discussion above. It takes as input the sets $C_1, \ldots, C_m$ and the color sequence $CS = (c_1, \ldots, c_m)$ and outputs the set C_1, where each point of C_1 now has a weight as defined above. The pseudocode for the algorithm is given as Algorithm 1. (In the pseudocode, if p has color c_i, then $nbr(p)$ is the weighted nearest neighbor of p of color c_{i+1}.)

Algorithm 1 : Preprocessing

Input: Sets $C_1, \ldots, C_m$ and the color sequence $CS = (c_1, \ldots, c_m)$
Output: Set C_1, where each point of C_1 now has a weight as defined before.

```
1: Assign each point of C_m a weight of 0
2: for i = m − 1 down to 1 do
3:    Build a weighted nearest neighbor search structure WNN_{i+1}, for the weighted
      points of C_{i+1}.
4:    for each point p ∈ C_i do
5:       nbr(p) ← weighted nearest neighbor of p of color c_{i+1} found by doing a weighted
         nearest neighbor query with p in WNN_{i+1}.
6:       w(p) ← weighted distance of p to nbr(p)
7:    end for
8: end for
```

The time complexity of Algorithm 1 is dominated by the time to build and query WNN_{i+1} for $i = m-1, \ldots, 1$ (line 3 and 5). We use our weighted Voronoi Diagram-based method (described in Section 2.4.3) to build and query WNN_{i+1}, which uses $O(n_{i+1})$ space and can be built in $O(n_{i+1} \log n_{i+1})$ time. Each of the n_i weighted nearest neighbor queries on WNN_{i+1} take $O(\log n_{i+1})$ time. Therefore, the total time is

$\sum_{i=1}^{m-1} O(n_{i+1} \log n_{i+1} + n_i \log n_{i+1}) = O(n \log n)$. The space complexity of the algorithm is also dominated by the space required for the WNN_{i+1}. Therefore, the space complexity of Algorithm 1 is $O\left(\max_{1 \le i \le m-1} \{n_{i+1}\}\right) = O(n)$, as the space can be reused.

2.3.2 An example

We illustrate the preprocessing step using the point-set in Figure 2.1, which has $m = 4$ colors and color sequence CS =(blue, red, magenta, green). We initialize the weight of each green point g_i to 0 (Figure 2.2a). Next we compute and assign to each magenta point m_i a weight equal to the distance to its weighted nearest green neighbor (Figure 2.2b). Then we compute and assign to each red point r_i a weight equal to the distance to its weighted nearest magenta neighbor (Figure 2.2c). Finally, we compute and assign to each blue point b_i a weight equal to the distance to its weighted nearest red neighbor (Figure 2.2d). With each point of each color, we also store a pointer to its weighted nearest neighbor of the next color in the sequence.

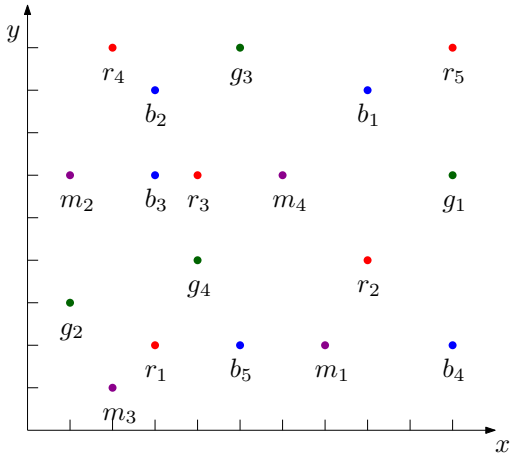
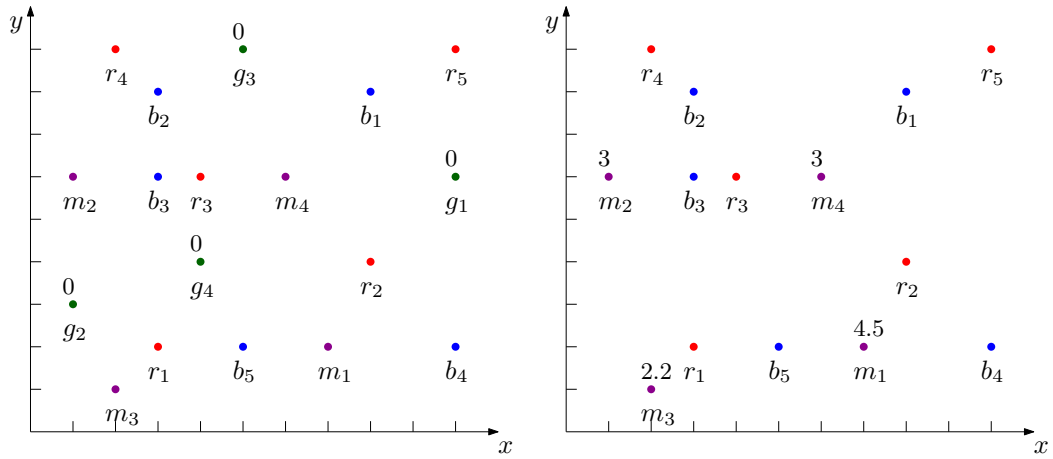


Figure 2.1: A point-set with 4 colors.

By Lemma 2.1, we can now answer an NNCTQ query on the original point-set with query point q by finding the weighted nearest blue neighbor of q. For instance, for the point q shown in Figure 2.2d, the weighted nearest blue neighbor is b_5 with weighted distance $d^w(q, b_5) = d(q, b_5) + w(b_5) = 7.8$. By tracing pointers back from b_5, we find that the optimal solution for q is the sequence (b_5, r_1, m_3, g_2).



(a) Weights of green points initialized to 0.

(b) Computed weight of magenta points. (Green points have been removed.)

(c) Computed weight of red points. (Magenta points have been removed.)

(d) Computed weight of blue points. (Red points have been removed.) The query point q is also shown.

Figure 2.2: Transformation from the multi-colored point-set of Figure 2.1 to a weighted point-set

2.4 Weighted Nearest Neighbor

Recall that we have transformed the NNCTQ problem on $S = C_1 \cup \cdots \cup C_m$, with query point q, to that of finding a point $b \in C_1$ that minimizes $d^w(q,b) = d(q,b) + w(b)$, where $w(b)$ is the weight of b, as assigned by Algorithm 1. We now present three algorithms, with different performance characteristics, for this problem. Section 2.4.1-2.4.3 describe, respectively, a threshold algorithm, a transformation-based algorithm, and a weighted Voronoi Diagram-based algorithm.

2.4.1 Threshold algorithm

This algorithm considers the points $b \in C_1$ in non-decreasing order of their Euclidean distance (not weighted distance) from q. Throughout its execution, the algorithm maintains the current best estimate of the desired minimum weighted distance, called the *threshold distance* d^w_{min}; initially $d^w_{min} = \infty$. For the current point, b, under consideration, if $d(q,b) \leq d^w_{min}$, then d^w_{min} is reset to the smaller of d^w_{min} and $d^w(q,b) = d(q,b) + w(b)$. Otherwise, the search terminates immediately with d^w_{min} equal to the desired minimum weighted distance. The correctness of this method follows from the fact that (i) the points $b \in C_1$, are considered in non-decreasing order of $d(q,b)$, and (ii) if $d(q,b) > d^w_{min}$ then $d^w(q,b) = d(q,b) + w(b) > d^w_{min}$ too (since $w(b) \geq 0$). Thus, no future point can reduce d^w_{min}.

The pseudocode for this algorithm is given as Algorithm 2. In this algorithm, the function $NN(q)$ returns the Euclidean nearest neighbor of q in C_1 and, in each iteration of the while-loop, $nextNN(q)$ returns the nearest neighbor of q amongst the points of C_1 that have not already been returned by previous calls to $NN(q)$ or $nextNN(q)$. Both functions can be implemented using a R-tree and a min-heap, similar to the k-nearest neighbor algorithm presented by Hjaltason *et al.* [68].

The space used by the query algorithm is dominated by the storage needs of the kd-tree, which is $O(|C_1|) = O(n)$. The query time depends on the configuration of q and the points of C_1. In the best case, the Euclidean distance from q to the second nearest neighbor may exceed the current threshold (i.e., the weighted distance to the nearest neighbor), which causes the algorithm to terminate after just one iteration of the while-loop. In this case, the query time is dominated by the query time of the

Algorithm 2 : Threshold-based method

Input: A set, C_1, of weighted points and query point q
Output: The point in C_1 with minimum weighted distance to q

```
1: d^w_min ← ∞
2: b ← NN(q)
3: while d(q, b) ≤ d^w_min do
4:    d^w_tmp ← d(q, b) + w(b)
5:    if d^w_tmp < d^w_min then
6:       d^w_min ← d^w_tmp
7:       wnn ← b      // wnn is the current weighted nearest neighbor
8:    end if
9:    b ← nextNN(q)
10: end while
11: return wnn
```

initial nearest neighbor query, which can be as low as $O(\log n)$. In the worst case, the Euclidean distance computed in each iteration is no larger than the current threshold, which causes the algorithm to check all points of C_1. In this case, the query time is proportional to n times the cost of a $nextNN(q)$ query (the latter can be as high as $O(n)$). In practice though, we expect the algorithm to perform quite well.

2.4.1.1 An example

Figure 2.3 shows a set $C_1 = \{b_1, \ldots, b_5\}$ of weighted points and a query point q. At the beginning of the algorithm, d^w_{min} is initialized to ∞. We first find the nearest neighbor of q, which in this case is b_2. Since $d(q, b_2) = 2 < d^w_{min}$, we compute $d^w_{tmp} = d^w(q, b_2) = 2 + 2.8 = 4.8$. Now since $d^w_{tmp} < d^w_{min}$, the new value of $d^w_{min} = d^w_{tmp} = 4.8$. Next, the algorithm computes the next nearest neighbor of q, which is b_3. Since, $d(q, b_3) = 2.8 < d^w_{min}$, we compute $d^w_{tmp} = d^w(q, b_3) = 2.8 + 1 = 3.8$. Since $d^w_{tmp} < d^w_{min}$, we update the new value of d^w_{min} to $d^w_{tmp} = 3.8$. The next nearest neighbor of q is b_1 with $d(q, b_1) = 3.6 < d^w_{min}$, so we compute $d^w_{tmp} = d^w(q, b_1) = 3.6 + 2 = 5.6$. Since the value of $d^w_{tmp} > d^w_{min}$ the threshold distance is not updated and the algorithm proceeds to find the next nearest neighbor which is b_4, with $d(q, b_4) = 6.4$. At this point the condition $d(q, b_4) \leq d^w_{min}$ fails and the algorithm terminates and returns b_3 as the weighted nearest neighbor, with weighed distance = 3.8.

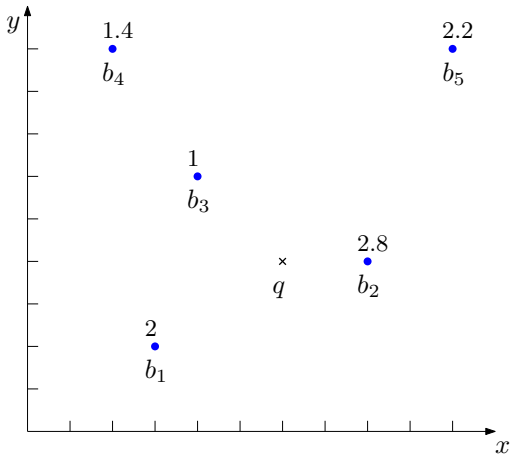


Figure 2.3: An example of a weighted point-set. Here q is the query point.

2.4.2 Transformation-based algorithm

In this algorithm, we transform the set C_1 of weighted points in $\mathbb{R}^2$ to a set C_1' of unweighted points in $\mathbb{R}^3$ and also transform the query point q in $\mathbb{R}^2$ to a suitable point q' in $\mathbb{R}^3$. We show that by finding the Euclidean nearest neighbor of q' in C_1' and mapping this point back to C_1 we get a provably-good approximation to the Euclidean weighted nearest neighbor of q in C_1.

2.4.2.1 The transformation

Let $b_i = (x(b_i), y(b_i))$ be any point in C_1, with weight $w(b_i)$. Let $q = (x(q), y(q))$ be the query point. We associate with q a "weight" $w(q) = 0$. The transformation maps b_i to the point $b_i' = (x(b_i), y(b_i), w(b_i))$ in $\mathbb{R}^3$ and maps q to the point $q' = (x(q), y(q), 0)$ in $\mathbb{R}^3$. Let $C_1' = \{b_i' | b_i \in C_1\}$.

2.4.2.2 A provably-good approximation

The following lemma establishes a relationship between the point in C_1 corresponding to the Euclidean nearest neighbor of q' in C_1' and the Euclidean weighted nearest neighbor of q in C_1.

Lemma 2.2. *Let b_i' be the point of C_1' that minimizes $d(q', b_i')$ and let b_k be the point of C_1 that minimizes $d^w(q, b_k) = d(q, b_k) + w(b_k)$. Then $d^w(q, b_i) \leq \sqrt{2} \cdot d^w(q, b_k)$.*

Proof. We have $d(q', b_i') = (x(q) - x(b_i))^2 + (y(q) - y(b_i))^2 + w(b_i)^2 = d(q, b_i)^2 + w(b_i)^2$. Similarly, $d(q', b_k') = d(q, b_k)^2 + w(b_k)^2$. Since, b_i' minimizes $d(q', b_i')$, we have

$$d(q, b_i)^2 + w(b_i)^2 \leq d(q, b_k)^2 + w(b_k)^2 \tag{2.1}$$

On the other hand, since b_k minimizes $d^w(q, b_k) = d(q, b_k) + w(b_k)$, we have

$$d(q, b_k) + w(b_k) \leq d(q, b_i) + w(b_i) \tag{2.2}$$

For notational convenience, let $D_i = d(q, b_i)$, $D_k = d(q, b_k)$, $w_i = w(b_i)$, and $w_k = w(b_k)$. Our goal is to compute an upper bound on $(D_i + w_i)/(D_k + w_k)$ subject to inequalities (2.1) and (2.2). Consider four non-negative real values D_i, w_i, D_k, and w_k satisfying inequalities (2.1) and (2.2). Let Z be a disk of radius $r = (D_k^2 + w_k^2)^{1/2}$ centered at the origin. The reals D_k and w_k can be represented as a point (D_k, w_k) on the boundary of Z. By inequality (2.1), D_i and w_i will then be represented by a point (D_i, w_i) in the interior or on the boundary of Z. (See Figure 2.4)

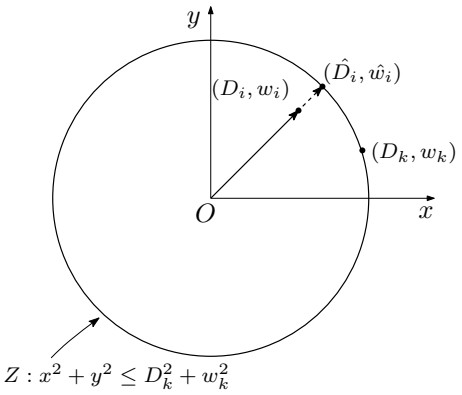


Figure 2.4: Disk Z with center at $(0, 0)$ and radius $r = (D_k^2 + w_k^2)^{1/2}$. Any point (D_i, w_i) satisfying inequality (2.1) is in the interior or on the boundary of Z.

Let $(\hat{D}_i, \hat{w}_i)$ be the point that is at the intersection of the radial line through (D_i, w_i) and the boundary of Z. Since $\hat{D}_i + \hat{w}_i \geq D_i + w_i$ and $(\hat{D}_i, \hat{w}_i)$ is on the boundary of Z, inequalities (2.1) and (2.2) hold for the reals $\hat{D}_i, \hat{w}_i, D_k$, and w_k. Straightforward calculations then show that the maximum value of $\hat{D}_i + \hat{w}_i$ is $\sqrt{2} \cdot r$ (achieved if $(\hat{D}_i, \hat{w}_i)$ coincides with the point $(r/\sqrt{2}, r/\sqrt{2})$ on the boundary of Z) and that the minimum value of $D_k + w_k$ is r (achieved if (D_k, w_k) coincides with the point $(r, 0)$ or $(0, r)$). Thus, $(D_i + w_i)/(D_k + w_k) \leq (\hat{D}_i + \hat{w}_i)/(D_k + w_k) \leq (\sqrt{2} \cdot r)/r = \sqrt{2}$. □

2.4.2.3 The query algorithm

The points of C_1' are preprocessed into a data structure for Euclidean nearest neighbor queries. Two candidates for this are the Voronoi Diagram in $\mathbb{R}^3$ and an R-tree. For the Voronoi Diagram approach, a query is answered by locating $q' = (x(q), y(q), 0)$ in a cell of the Voronoi Diagram, VD, and reporting the associated generator. We can simplify the point location phase by observing that since the w-coordinate of q' is zero, we can build the point location structure on the 2-dimensional subdivision obtained by intersecting VD with the plane $w = 0$ and locating $q = (x(q), y(q))$ in this. Since the size of the Voronoi Diagram in $\mathbb{R}^3$ is $O(n^2)$, explicitly computing VD will require $O(n^2)$ space. Therefore, we will implicitly compute the subdivision by considering the locus of points only in $w = 0$ plane. The overall storage required for this method is $O(n)$ and the query time is $O(\log n)$.

Alternatively, as discussed earlier, the R-tree approach takes $O(n)$ space. The query time ranges from $O(\log n)$ to $O(n)$ depending on the configuration of the input points and the query point. However, in practice we expect this method to perform quite well.

2.4.2.4 Exact solution for L_1-distance metric

Interestingly, the transformation-based method yields the true weighted nearest neighbor of q in C_1 if the Manhattan distance (i.e., L_1 distance) metric is substituted for the Euclidean distance (i.e., L_2 distance) metric, as we will show now. Throughout this section, we take $d(\cdot, \cdot)$ to be the L_1-distance function.

Lemma 2.3. *A point $b_i \in C_1$ with a weight $w(b_i)$ is the weighted nearest neighbor of query point q under the L_1-distance metric iff point $b_i' \in C_1'$ is the nearest neighbor of q' under the L_1-distance metric.*

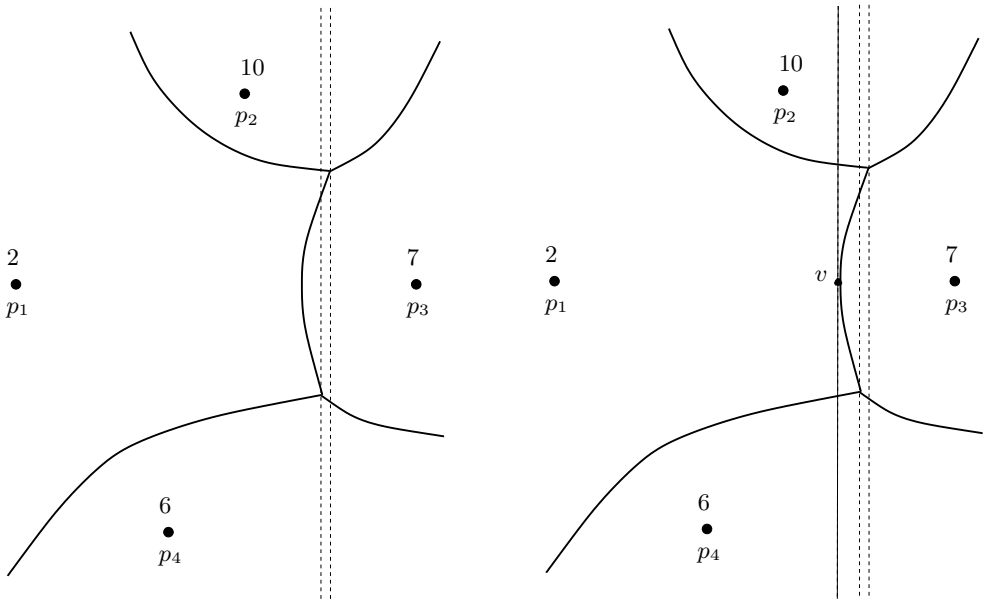
Proof. We have $d^w(q, b_i) = d(q, b_i) + w(b_i)$ is minimum iff $|x(q) - x(b_i)| + |y(q) - y(b_i)| + w(b_i)$ is minimum, i.e., iff $|x(q) - x(b_i)| + |y(q) - y(b_i)| + |0 - w(b_i)|$ is minimum, i.e., iff $d(q', b_i')$ is minimum. □

The only change needed to the query algorithm is that the points of C_1' are preprocessed into a data structure for L_1-distance nearest neighbor queries. For this we can either build an L_1-distance Voronoi Diagram in $\mathbb{R}^3$ or build an R-tree and use

L_1-distances to answer the nearest neighbor query in $\mathbb{R}^3$. Similar to the L_2-distance metric, the overall storage required for both methods is $O(n)$. The query time for Voronoi diagram-based approach is $O(\log n)$. The query time for the R-tree approach ranges from $O(\log n)$ to $O(n)$ depending on the configuration of the input points and the query point.

2.4.3 Weighted Voronoi Diagram-based solution

In this section, we present an additively weighted Voronoi Diagram-based solution for the weighted nearest neighbor problem. Recall the Voronoi Diagram-based approach for nearest neighbor search discussed in Section 2.2. We use a similar approach for the weighted nearest neighbor query on C_1, except that now a additively weighted Voronoi Diagram $\mathcal{V}_w(C_1)$ is used. However, care must be taken in using the point location structure of [66].



(a) There is no total order on the segments in the leftmost strip.

(b) Adding a new vertex v yields a total ordering of the segments in each strip.

Figure 2.5: Weighted Voronoi Diagram for a set of 4 weighted points. (Adapted from [62].)

The presence of hyperbolic arcs in $\mathcal{V}_w(C_1)$ can lead an absence of a total order among

segments in some strips, as seen in Figure 2.5a. Fortunately, this can be rectified by dividing each hyperbolic arc into two x-monotone sub-arcs and adding a vertex at the leftmost or rightmost point of the arc. These added vertices generate additional vertical strips and within each strip there is now a total order among the segments, as seen in Figure 2.5b.

As $\mathcal{V}_w(C_1)$ has $O(|C_1|) = O(n)$ hyperbolic arcs, this introduces only $O(n)$ new vertices. Hence the point location structure occupies $O(n)$ space and answers queries in $O(\log n)$ time. Therefore, the weighted nearest neighbor problem can be solved by using a weighted Voronoi Diagram VD_1 on the weighted points in C_1 and a point location data structure on VD_1. Given q, we locate it in VD_1 and report the point of C_1 that is the generator of the cell containing q. This method uses $O(|C_1|) = O(n)$ space and $O(n \log n)$ preprocessing time to build a data structure that supports queries in $O(\log n)$ time.

Chapter 3

Colored Tuple Range Query (CTRQ)

The goal of the range search problem is to preprocess a given set of points so that the points lying inside a query object (e.g., a rectangle, or a ball, or a halfspace) can be reported efficiently. Due to its many applications in diverse domains such as computer graphics, robotics, database systems, etc., the range search problem has been studied extensively in the computational geometry and the database literature and space-and query time-efficient solutions have been devised for many instances of the problem (see, for instance [2, 3, 7, 8, 9]).

In this chapter we consider the point-tuple version of the circular range search problem (Section 1.2.2). We present efficient solutions for the fixed distance and the variable distance versions of this problem, where the distance constraint (δ) is known beforehand or is specified as part of the query, respectively.

3.1 Problem formulation

Problem 3.1 (CTRQ)**.** *Let S be a set of n points in $\mathbb{R}^2$, where each point is assigned a color c_i from a palette of m colors ($m \geq 2$). Let C_i be the set of points of color c_i and let $n_i = |C_i|$. Note that $C_i \cap C_j = \emptyset$ if $i \neq j$ and $\bigcup_{i=1}^{m} C_i = S$. Let $CS = (c_1, \ldots, c_m)$ be a given ordering of the colors. We wish to preprocess S into a suitable data structure so that for any query point q in $\mathbb{R}^2$ and a distance $\delta > 0$, we can report all tuples*

$(p_1, \ldots, p_m)$, where $p_i \in C_i$, such that $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i) \leq \delta$. (Here $d(\cdot, \cdot)$ denotes Euclidean distance.)

Throughout, we will discuss the CTRQ problem for $m = 2$ colors, which will henceforth be called the 2-CTRQ problem. As we will discuss in Section 3.4, our solutions for 2-CTRQ generalize to $m > 2$ colors. For simplicity, we define the 2-CTRQ problem as follows.

Problem 3.2 (2-CTRQ)**.** *Let B be a set of blue points, $B = \{b_1, b_2 \ldots, b_{n_1}\}$, and R be a set of red points, $R = \{r_1, r_2 \ldots, r_{n_2}\}$, in $\mathbb{R}^2$, where $n_1 + n_2 = n$. We wish to preprocess the sets, B and R, so that for any query point $q : (x_q, y_q) \in \mathbb{R}^2$ and distance $\delta > 0$, we can report all tuples (b_i, r_j), such that $d(q, b_i) + d(b_i, r_j) \leq \delta$.*

3.1.1 Approach and key ideas

Our algorithms employ a 2-step approach. In the first step, we identify an appropriate subset of blue points from which to search for (blue, red) pairs to output (with respect to the query point, q, and query distance δ). In the second step we search from each such blue point and identify those red points that are reachable from q within distance δ and output the corresponding (blue, red) pairs.

The identification of blue points in the first step needs to be done with care. Selecting too few of these will result in some valid (blue, red) pairs not being reported. On the other hand, selecting too many will lead to a high query time since many of them may not be part of any reported pair (as the CTRQ example in Section 1.2.2 shows).

Thus, a key idea underlying our algorithms is to select efficiently *precisely* those blue points that are guaranteed to be part of at least one reported (blue, red) pair, with respect to the query point, q, and query distance δ. This, combined with the second step above, ensures that only those pairs that should be reported are actually reported. Moreover, time is not wasted in processing blue points that will not contribute to the output. We will refer to the blue points so selected as *fruitful* points.

A second key idea concerns tuning the query algorithm to the (unknown) output size. In general, a straightforward application of the 2-step method above does not necessarily result in the best possible query time, particularly if the output size is "small" since then the overall query time is dominated by other terms in the query time. To help

balance these costs, we show how to identify a suitable threshold on the output size as a function of the input size, n, and how to query judiciously on the two sides of the threshold to achieve the desired balance; however, this improvement usually comes at the expense of more space. (We note that this is just a rough description of the approach and the general idea manifests itself in different ways in the fixed and the variable distance problems.)

3.2 Fixed distance CTRQ

In this section, we present our solution for the fixed distance version of the 2-CTRQ problem (Problem 3.2). Note that in the fixed distance version, δ is known beforehand (during preprocessing). We first describe an algorithm that uses $O(n)$ space and has a query time of $O((k+1)\log n)$, where k is the output size. We then show how to generalize this to a space-query time tradeoff.

As mentioned in Section 3.1.1, the notion of fruitful blue points is central to our approach. Recall that a blue point is a fruitful point if it is part of at least one output tuple for a given q and δ. The following lemma provides a useful characterization of such points.

Lemma 3.1. *For any $b_i \in B$, let $r_j \in R$ be its closest red point. Let q be a query point and δ the (fixed) query distance. Then b_i is fruitful if and only if $d(q,b_i)+d(b_i,r_j) \leq \delta$.*

Proof. ($\Leftarrow$) If $d(q,b_i)+d(b_i,r_j) \leq \delta$ then the tuple (b_i,r_j) is output. Hence, by definition, b_i is fruitful.

($\Rightarrow$) Suppose that b_i is fruitful. Thus, there is an $r_k \in R$ such that (b_i,r_k) is output. Thus, $d(q,b_i)+d(b_i,r_k) \leq \delta$. Since $d(b_i,r_j) \leq d(b_i,r_k)$, we have $d(q,b_i)+d(b_i,r_j) \leq \delta$. □

In other words, b_i is fruitful if (b_i,r_j) is part of the answer set, and not fruitful otherwise. Note that whether or not b_i is fruitful (for a fixed δ) depends on q, i.e., b_i may be fruitful for some q and may not fruitful for some other q.

Lemma 3.1 suggests the following preprocessing step to help determine the fruitful point at query time. For each $b_i \in B$, we compute its nearest red point $r_j \in R$ and associate with b_i the real number $d(b_i,r_j)$ as a *weight* $w(b_i)$. For a query point q, b_i is

fruitful if and only if $d(q, b_i) + w(b_i) \leq \delta$, i.e., $d(q, b_i) \leq \delta - w(b_i)$, i.e., if and only if q lies inside the disk D_i of radius $\delta - w(b_i)$ centered at b_i. Thus, we build a data structure on the disks D_i that can identify the disks that are "stabbed" by q, and hence can identify the fruitful blue points. This facilitates the first step of our query algorithm.

The second step is to identify for each fruitful blue point, b_i, all the red points r_k (not merely the nearest red point) that are reachable from q within a distance of δ. Each such r_k satisfies the condition $d(q, b_i) + d(b_i, r_k) \leq \delta$, i.e., $d(b_i, r_k) \leq \delta - d(q, b_i)$, i.e., r_k is in a disk of radius $\delta - d(q, b_i)$ centered at b_i. To facilitate the identification of such red points, r_k, we build a data structure for circular range queries on the set, R, of red points.

We argue that a tuple (b_i, r_j) is reported if and only if $d(q, b_i) + d(b_i, r_j) \leq \delta$. Suppose that (b_i, r_j) is reported. Thus, r_j is found when DS_t is queried with b_i and radius $\rho_i = \delta - d(q, b_i)$, i.e., $d(b_i, r_j) \leq \delta - d(q, b_i)$, i.e., $d(q, b_i) + d(b_i, r_j) \leq \delta$. On the other hand, suppose that $d(q, b_i) + d(b_i, r_j) \leq \delta$. Then, $d(q, b_i) + d(b_i, nn_i) \leq \delta$, where nn_i is the red point nearest to b_i. Thus, by Lemma 3.1, b_i is a fruitful point. Hence DS_t is queried with b_i and radius $\rho_i = \delta - d(q, b_i) \geq \delta - (\delta - d(b_i, r_j)) = d(b_i, r_j)$, which implies that r_j is found by the query. Hence, (b_i, r_j) is reported. This establishes the correctness of the query algorithm.

The above preprocessing steps are shown as Algorithm 6.

As noted in Section 1.3.2, circular range search in $\mathbb{R}^2$ can be transformed to halfspace range search in $\mathbb{R}^3$ via the lifting map [34]. Likewise, disk stabbing can also be transformed to halfspace range search in $\mathbb{R}^3$ using the lifting map followed by geometric duality [34]. Therefore, the data structure for the disk stabbing DS_f and the circular range search DS_t can be implemented as the halfspace range search structures developed in [35]. They occupy $O(n)$ space and answer queries in $O(\log n + k)$ time, where k is the output size. Additionally, the nearest neighbor-finding structure (Voronoi diagram and point location structure) can be implemented in $O(n)$ space and has a query time of $O(\log n)$ [2].

The query algorithm is shown as Algorithm 7. The first step identifies the set F of blue points by querying DS_f and the second step identifies tuples to output by querying DS_t.

We argue that a tuple (b_i, r_j) is reported if and only if $d(q, b_i) + d(b_i, r_j) \leq \delta$.

Algorithm 3 : Preprocessing

Input: B, R: set of input points; δ: a real

Output: DS_f: Data structure to find fruitful points; DS_t: Data structure to form tuples.

```
1: D ← ∅ // set of disks
2: Create Voronoi diagram, V_r, for points in R.
3: Build a point location structure on V_r.
4: for all b_i ∈ B do
5:    nn_i ← nearest neighbor red point of b_i via point location in V_r.
6:    if d(b_i, nn_i) ≤ δ then // b_i cannot be fruitful if d(b_i, nn_i) > δ
7:       w(b_i) ← d(b_i, nn_i)
8:       D_i ← disk with radius δ − w(b_i) and centered at b_i
9:       D ← D ∪ {D_i}
10:   end if
11: end for
12: Create disk stabbing data structure, DS_f, on disks in D
13: Create circular range search data structure, DS_t, on points in R
```

Suppose that (b_i, r_j) is reported. Thus, r_j is found when DS_t is queried with b_i and radius $\rho_i = \delta - d(q, b_i)$, i.e., $d(b_i, r_j) \leq \delta - d(q, b_i)$, i.e., $d(q, b_i) + d(b_i, r_j) \leq \delta$. On the other hand, suppose that $d(q, b_i) + d(b_i, r_j) \leq \delta$. Then, $d(q, b_i) + d(b_i, nn_i) \leq \delta$, where nn_i is the red point nearest to b_i. Thus, by Lemma 3.1, b_i is a fruitful point. Hence DS_t is queried with b_i and radius $\rho_i = \delta - d(q, b_i) \geq \delta - (\delta - d(b_i, r_j)) = d(b_i, r_j)$, which implies that r_j is found by the query. Hence, (b_i, r_j) is reported. This establishes the correctness of the query algorithm.

The first step takes $O(\log n + k_F) = O(\log n + k)$ time, where $k_F = |F|$ is the number of fruitful points and k is the size of the output for the 2-CTRQ query. Crucially, note that $k_F \leq k$ since, by definition, every fruitful point belongs to at least one output tuple. The second step takes $O(\log n + k_i)$ time for each $b_i \in F$, where k_i is the size of output of the circular range search from b_i, i.e., the number of red points reported, hence the number of tuples that b_i is a part of in the answer set. Therefore, the total time for the second step is $O(\sum_{b_i \in F}(\log n + k_i)) = O((k+1)\log n)$ since $\sum_{b_i \in F} k_i = k$. Thus, the second step dominates the overall query time, which is $O((k+1)\log n)$. Each of the structures $\mathcal{V}_r$, DS_f, and DS_t uses $O(n)$ space, so the total space used is $O(n)$.

Algorithm 4 : Query

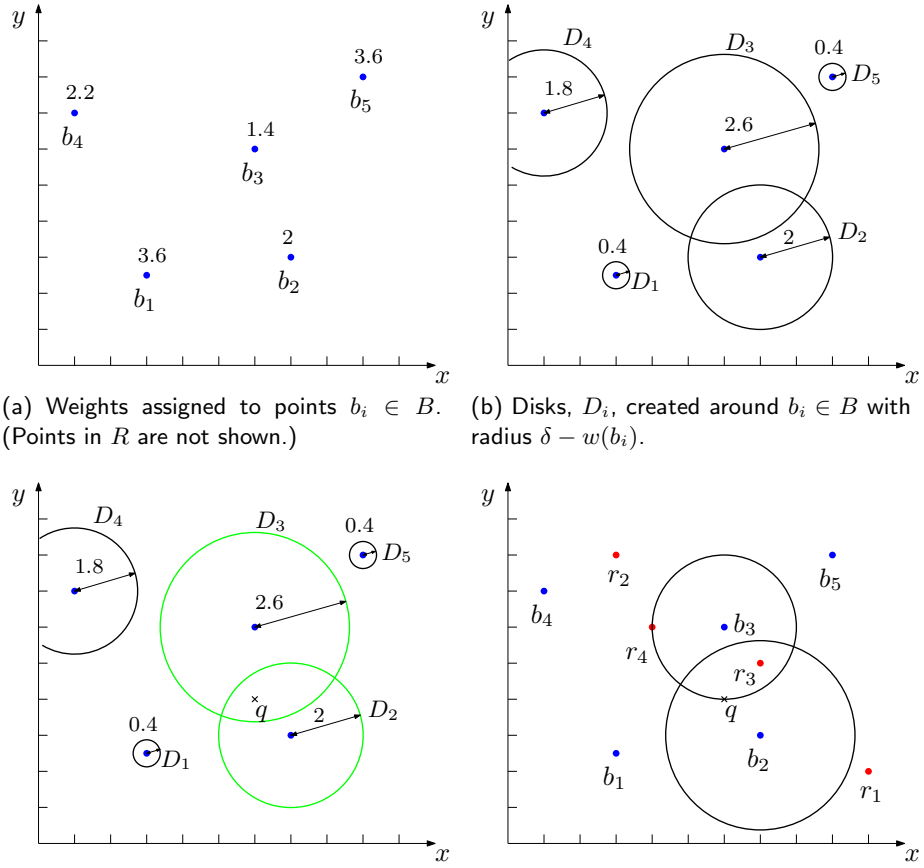
Input: DS_f: Data structure to find fruitful points; DS_t: Data structure to form tuples; δ: a real; q: query point

Output: O: set of colored tuples satisfying the distance constraint δ with respect to q

1: $O \leftarrow \emptyset$
   // `Find fruitful points for` q
2: $D' \leftarrow$ Subset of disks D stabbed by q, as found by a disk stabbing query on DS_f with q
3: $F \leftarrow$ Subset of blue points that are centers of disks in D' // F `is the set of fruitful blue points`
4: **for all** $b_i \in F$ **do**
5: $\quad \rho_i \leftarrow \delta - d(q, b_i)$
   $\quad$ // `Query using fruitful points to form tuples`
6: $\quad R'_i \leftarrow$ Subset of red points found by a circular range search on DS_t with b_i as query point and radius ρ_i
7: $\quad$ **for all** $r_j \in R'_i$ **do**
8: $\quad\quad O \leftarrow O \cup \{(b_i, r_j)\}$
9: $\quad$ **end for**
10: **end for**
11: **return** O

3.2.1 An example

We illustrate our algorithm using the point-set of Figure 1.3 with $\delta = 4$. First, for each point $b_i \in B$, we compute its nearest neighbor in R and assign the distance as weight $w(b_i)$ to b_i as shown in Figure 3.1a. Next, we filter out all the b_i's for which $w(b_i)$ is larger than δ and create a disk, D_i, around each remaining weighted point $b_i \in B$ with radius $\delta - w(b_i)$, as shown in Figure 3.1b. (Since, in this example, we do not have any b_i with weight larger than δ, we do not discard any weighted points.) For a given query point, q, as shown in Figure 3.1c, we find the disks D_2 and D_3 (shown in green) stabbed by q. The points b_2 and b_3 corresponding to the centers of the stabbed disks D_2 and D_3, respectively, are the fruitful points. Next, as shown in Figure 3.1d, we perform two circular range search queries, one centered at b_2 with radius $\delta - d(q, b_2) = 4 - 1.4 = 2.6$, and one centered at b_3 with radius $\delta - d(q, b_3) = 4 - 2 = 2$. We then form tuples (b_2, r_3) with total distance 3.41 from q, since r_3 is the only point reported in the circular range search with b_2 as center, and (b_3, r_3) and (b_3, r_4) with total distance 3.41 and 4, respectively, from q, since r_3 and r_4 are the points reported in the circular range search



(a) Weights assigned to points $b_i \in B$. (Points in R are not shown.)

(b) Disks, D_i, created around $b_i \in B$ with radius $\delta - w(b_i)$.

(c) Disks found to be stabbed by q (shown in green).

(d) Tuples found via circular range search.

Figure 3.1: A running example to illustrate our solution for fixed distance 2-CTRQ with $\delta = 4$.

with b_3 as center.

3.2.2 Space-time trade-off

Ideally, one would like a query time of $O(\log n + k)$ for the 2-CTRQ problem instead of $O((k+1)\log n) = O(\log n + k \log n)$. The roadblock to this is the second step, i.e., the time to perform a circular range query from each fruitful blue point. For a fruitful point b_i, the circular range query takes $O(\log n + k_i)$ time, where k_i is the number of red points reported. An important observation is that if $k_i \geq \log n$ then $O(\log n + k_i) = O(k_i)$;

otherwise $O(\log n + k_i) = O(\log n)$. Based on this observation, we can divide the fruitful points into two sets; *heavy* fruitful points, b_i, for which $k_i \geq \log n$ and *light* fruitful points, b_i, for which $k_i < \log n$. The heavy fruitful points are, in fact, "good" points because the output size (k_i) dominates the query overhead ($O(\log n)$), which allows us to absorb the latter inside the former and obtain a query bound of $O(k)$ for such points. On the other hand, for the light fruitful points, the query overhead dominates the output size and, therefore, results in the $k \log n$ term in the query time.

Therefore, our approach is to use a simpler data structure for the light fruitful points that avoids the $O(\log n)$ query overhead. For the heavy fruitful points, we will continue to use the circular range search structure DS_t. Note that we do not know ahead of time which blue points are fruitful and which among these are light or heavy, as this depends on q. Therefore, we will build the simpler data structure for each blue point $b_i \in B$. This structure is merely a linked list, L_i, which contains the $(\log n)$-nearest red points to b_i, stored in non-decreasing order of their distances from b_i (ties broken arbitrarily).

To answer a query q, we first identify the set, F, of fruitful blue points using DS_f. Next, we classify each $b_i \in F$ as light or heavy as follows. Let $\hat{r_i}$ denote the last point in L_i, i.e., the one that is $(\log n)$-nearest from b_i. It is easy to see that if $d(q, b_i) + d(b_i, \hat{r_i}) \leq \delta$ then b_i is heavy; otherwise b_i is light. For each b_i that is light, we scan L_i in non-decreasing order and output (b_i, r_j) for each r_j that is seen such that $d(q, b_i) + d(b_i, r_j) \leq \delta$. (We stop the scan as soon as this condition fails.) The query time is $O(k_i + 1)$, which, as desired, avoids the $O(\log n)$ query overhead that was incurred previously. Thus, the total query time for all light points queried is $O(k + k_F) = O(k)$, since $k_F \leq k$. For each heavy point b_i, we query DS_t, which takes $O(\log n + k_i) = O(k_i)$ time (since $k_i \geq \log n$). Thus the query time for all heavy points queried is $O(k)$. Thus the total time for the second step of the 2-CTRQ query is $O(k)$. The overall query time, inclusive of the first step, is hence $O(\log n + k)$, which improves upon the bound of $O((k+1) \log n)$ obtained previously. This improvement comes at the expense of a slightly higher space bound of $O(n \log n)$ since we store a list L_i of length $\log n$ with each $b_i \in B$.

The preceding approach can be generalized. For a user-specified integer parameter t, $1 \leq t \leq \log n$, we store a list of the t-nearest red points with each blue point. For a given q, there can be at most $\frac{k}{t}$ heavy fruitful points, where k is the output size, since the output size of a heavy blue point is, by definition, at least t. Therefore, we

perform at most $\frac{k}{t}$ circular range queries, each of which takes $O(\log n + k_i)$ time and yields a total of $O(\frac{k}{t}\log n + k) = O(\frac{k}{t}\log n)$ over all heavy points. The time for the light points continues to be $O(k)$, as before. Therefore, the overall time for the 2-CTRQ query, inclusive of the first step, is $O(\log n + \frac{k}{t}\log n + k) = O((1 + \frac{k}{t})\log n)$ and the space requirement is $O(nt)$. For example, if $t = \sqrt{\log n}$ then we get a scheme with $O(n\sqrt{\log n})$ space and $O(\log n + k\sqrt{\log n})$ query time.

Theorem 3.1. *A fixed-distance* 2-CTRQ *problem on a set of n red and blue points can be solved in $O((1 + \frac{k}{t})\log n)$ time using $O(nt)$ space, where t is a user-specified parameter, $1 \leq t \leq \log n$ and k is the output size. In particular, $t = 1$ (resp. $t = \log n$) yields a solution with $O((k+1)\log n)$ (resp. $O(\log n + k)$) query time and $O(n)$ (resp. $O(n \log n)$) space.*

3.3 Variable distance CTRQ

In this section, we discuss our solution for the variable distance 2-CTRQ problem (Problem 3.2). As before, we present a two-step solution for this problem, where, in the first step, we find the set of fruitful points and, in the second step, we explore B and R using the fruitful points to form and report the tuples. However, the challenge now is that δ is not known at preprocessing time. Therefore it is not possible to build the disk stabbing data structure DS_f to find the fruitful blue points. In the rest of this section we discuss alternative methods to find these fruitful points. (The second step of our query algorithm remains unchanged as it does not rely on knowing δ at preprocessing time.)

It is easy to see that the approach underlying the Lemma 3.1 continues to hold even if δ is variable. Thus, as before, we precompute for each $b_i \in B$ its closest red point $r_j \in R$ and store with b_i the weight $w(b_i) = d(b_i, r_j)$. Thus, the problem of finding fruitful points of B for q and δ boils down to finding a set of weighted points $b_i \in B$ such that $d(q, b_i) + w(b_i) \leq \delta$. In Section 3.3.1, we describe an algorithm for finding fruitful points that is based on a certain geometric transformation. Then in Section 3.3.2, we describe a method based on additively-weighted higher-order Voronoi diagrams to find the fruitful points. In Section 3.3.2.1 we show how to improve the storage requirement of the method of Section 3.3.2 using a graph traversal technique to

record certain information compactly. In Section 3.3.2.2 we discuss briefly how to also improve the query time by using a probabilistic method to suitably partition the points in preprocessing. (The query time remains deterministic.) Finally, in Section 3.3.3, we describe the overall algorithm for the variable distance 2-CTRQ problem.

3.3.1 Geometric transformation-based algorithm for fruitful points

The main idea behind this approach is to transform the weighted blue points in $\mathbb{R}^2$ to (unweighted) points in $\mathbb{R}^4$ in such a way that the solution to a certain half-space range search problem on these points in $\mathbb{R}^4$ yields the desired fruitful points in $\mathbb{R}^2$. We propose the following transformation:

1. Point $b_i = (x_i, y_i)$ in $\mathbb{R}^2$, with weight $w(b_i)$ is mapped to point $b_i' = (x_i, y_i, w(b_i), z_i)$ in $\mathbb{R}^4$, where $z_i = x_i^2 + y_i^2 - w(b_i)^2$.

2. Query point $q = (x_q, y_q)$ in $\mathbb{R}^2$ with query distance δ is mapped to hyperplane q' in $\mathbb{R}^4$ defined by the equation $a_1 x + a_2 y + a_3 w + a_4 z = c$, where $a_1 = -2x_q, a_2 = -2y_q, a_3 = 2\delta, a_4 = 1$, and $c = \delta^2 - x_q^2 - y_q^2$.

The following lemma establishes the desired property of this transformation.

Lemma 3.2. *Consider the transformation given above. Then, for any $b_i \in B$, $d(q, b_i) + w(b_i) \leq \delta$ if and only if b_i' is on or below q'.*

Proof. We have

$$
\begin{aligned}
& d(q, b_i) + w(b_i) \leq \delta \\
\Leftrightarrow \quad & \sqrt{(x_q - x_i)^2 + (y_q - y_i)^2} \leq \delta - w(b_i) \\
\Leftrightarrow \quad & x_q^2 + y_q^2 + x_i^2 + y_i^2 - 2x_q x_i - 2y_q y_i \leq \delta^2 + w(b_i)^2 - 2\delta w(b_i) \\
\Leftrightarrow \quad & (-2x_q)x_i + (-2y_q)y_i + (2\delta)w(b_i) + (x_i^2 + y_i^2 - w(b_i)^2) \leq \delta^2 - x_q^2 - y_q^2 \\
\Leftrightarrow \quad & a_1 x_i + a_2 y_i + a_3 w(b_i) + a_4 z_i \leq c \\
\Leftrightarrow \quad & b_i' \text{ is on or below } q'
\end{aligned}
$$

□

Based on Lemma 3.2, we transform the weighted blue points in $\mathbb{R}^2$ to points in $\mathbb{R}^4$ and create a data structure for half-space range search on these points in $\mathbb{R}^4$, let it be DS_{4hs}. Given q and δ, we compute q' and query the data structure with q' to find the points that are on or below q', hence the fruitful points. The query algorithm is shown as Algorithm 5.

Algorithm 5 : Geometric-Transformation-Based Method

Input: DS_{4hs}: Data structure for half space range search in $\mathbb{R}^4$; q: query point; δ: query distance
Output: F: set of fruitful blue points

1: $F \leftarrow \emptyset$
2: $q' \leftarrow$ hyperplane in $\mathbb{R}^4$ obtained by applying the above-mentioned transformation on q and δ
   // `Perform half-space range search`
3: $F' \leftarrow$ set of points returned when DS_{4hs} in queried with q'
4: **for all** $b'_i \in F'$ **do**
5: $\quad b_i \leftarrow$ point in $\mathbb{R}^2$ corresponding to b'_i
6: $\quad F \leftarrow F \cup \{b_i\}$
7: **end for**
8: **return** F

As mentioned in Section 1.3.2, for $d \geq 4$, there is a data structure for halfspace range search in $\mathbb{R}^d$ that has a query time of $O(n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n + k)$ and uses $O(n)$ space [35]. We use this structure for DS_{4hs}, with $d = 4$. Therefore, the space used by this approach is $O(n)$ and the query time is $O(\sqrt{n} \log^{O(1)} n + k_F)$.

Lemma 3.3. *Let B be a set of blue points in the plane, where $|B| \leq n$. The points of B that are fruitful with respect to a query point q and query distance δ can be computed in $O(\sqrt{n} \log^{O(1)} n + k_F)$ time and $O(n)$ space, where k_F is the number of fruitful points.*

3.3.2 Finding fruitful points using additively-weighted higher-order Voronoi diagrams

Recall that each of the k_F fruitful blue points, b_i, satisfies a distance constraint relative to q; specifically, $d(q, b_i) + w(b_i) \leq \delta$. Therefore, if the points of B are ordered by non-decreasing weighted distance from q, then the fruitful points are exactly the first k_F points in this order. In other words, the fruitful points constitute the set of k_F-closest

neighbors of q, under the weighted distance function.

Thus, we find fruitful points by finding the k_F-closest neighbors of q, under weighted distance. To accomplish this, we could build on the points of B an order-k_F additively-weighted Voronoi diagram in the plane [61]. This diagram partitions the plane into cells and associates with each cell a set of k_F blue points (called *generators*) that are the k_F-closest neighbors, under weighted distance, of any point $p \in \mathbb{R}^2$ lying in the cell. (The ordering of the generators can be different for different points p in the cell.) Thus, the generators associated with the cell containing q (determined via point location in the diagram) constitute the desired fruitful points.

Unfortunately, this approach is not feasible since q and δ are not known beforehand, hence k_F is not known at preprocessing time. Therefore, we build the following data structure. We create a complete binary tree, $\mathcal{T}$, on the points of B where at most $h = c \log n$ points, in any order, are stored at the leaves, for some constant $c \geq 1$. (In Section 3.3.2.2, we show that a more careful distribution of points can further improve the query time.) At each non-leaf node $v \in \mathcal{T}$ (including the root), we store an order-h additively-weighted Voronoi diagram $\mathcal{V}_h(v)$ built on the set, $B(v)$, of points stored at the leaves of the subtree rooted at v. We also build a data structure for doing point location in $\mathcal{V}_h(v)$.

Given q and δ, we explore $\mathcal{T}$, starting at the root, to determine the fruitful points as follows. At node v, if v is a leaf node then we compute the weighted distance from q to the points stored at v, and report the points with distance at most δ as fruitful points. Otherwise, we query $\mathcal{V}_h(v)$. This involves locating q in $\mathcal{V}_h(v)$, retrieving the h generators of the cell containing q, computing the weighted distance from q to each generator, and checking this distance against δ. If there is a generator with weighted distance larger than δ, then $k_v < h$, where k_v is the number of fruitful points in $B(v)$. We report the generators that have weighted distance at most δ from q as fruitful points and abandon searching below v. Otherwise, $k_v \geq h$. In this case, we do not report anything, but instead continue to explore the two children of v.

We analyze the query time as follows. For a given q and δ, each node v of $\mathcal{T}$ that is explored in the search is of one of two types: (i) v is a non-leaf node for which $k_v \geq h$. We do not report any fruitful points at v but instead explore v's children. We call v a *non-terminal* node. And (ii) v is a non-leaf for which $k_v < h$ or a leaf. We report

fruitful points (if any) at v and abandon the search below v. We call v a *terminal* node.

The time spent at a non-terminal node, for point location and for checking the generators, is $O(\log n + h) = O(\log n)$. Similarly, for the time spent at a terminal node that is not a leaf of $\mathcal{T}$. At a terminal node that is a leaf, we spend $O(h) = O(\log n)$ time. Therefore, to complete the query time analysis, we need to only upper-bound the number of terminal and non-terminal nodes.

The number of terminal nodes is at most twice the number of non-terminal nodes, since a terminal node that is not the root has a non-terminal node as parent and since $\mathcal{T}$ is binary. Therefore the total number of terminal and non-terminal nodes is at most three times the number of non-terminal nodes plus one. (The one extra node in the count handles the case where the root is the only terminal node in the search.)

To upper-bound the number of non-terminal nodes, consider any such node v. We charge one unit for v and distribute this charge uniformly over the $k_v \geq h$ fruitful points in v's subtree, which results in a charge of at most $1/h$ to each of these fruitful points. In the worst case, any fruitful point in $\mathcal{T}$ is charged at most $1/h$ in this way for each non-terminal node to whose subtree it belongs. Since the height of $\mathcal{T}$ is $O(\log n)$, this results in a total charge of $O((\log n)/h) = O(1)$ per fruitful point. It follows that the total number of terminal and non-terminal nodes is $O(1 + k_F)$. Thus, the query time is $O((1 + k_F) \log n)$.

The size of $\mathcal{V}_h(v)$ at a node v, inclusive of the h generators stored with each cell, is $O(h^2(|B(v)| - h)) = O(h^2|B(v)|)$ [61]. For nodes v at the same depth in $\mathcal{T}$, the sets $B(v)$ are pairwise disjoint. Therefore, the total size of $\mathcal{V}_h(v)$ at these nodes v is $O(h^2 n)$. Since the maximum depth in $\mathcal{T}$ is $O(\log n)$, the total space used is $O(h^2 n \log n) = O(n \log^3 n)$.

3.3.2.1 Reducing the space used

The space usage can be improved as follows. Note that the size of the symmetric difference of the sets of the generators associated with any two consecutive cells of a $\mathcal{V}_h(\cdot)$, is two, i.e., consecutive cells have $h - 1$ common generators. Based on this observation, we do not store the list of h generators with each cell; instead, we create a query data structure to find the list of generators for a cell. We first create a dual graph $\mathcal{G}$ of $\mathcal{V}_h(\cdot)$, where each vertex of $\mathcal{G}$ corresponds to a cell in $\mathcal{V}_h(\cdot)$ and two vertices in $\mathcal{G}$ are connected by an edge if and only if the corresponding cells in $\mathcal{V}_h(\cdot)$ share an

edge. Note that the number of vertices in $\mathcal{G}$ is the same as the number of cells in $\mathcal{V}_h(\cdot)$, which we denote by $|\mathcal{V}_h(\cdot)|$. Next, we create a spanning tree of $\mathcal{G}$ and duplicate each edge in it and perform an Euler tour on it. (An Euler tour exists because the resulting graph is undirected and all the vertices have even degree.)

We record the "time" when a vertex is visited during the tour using a sequence of integers $1, 2, \ldots, 2|\mathcal{V}_h(\cdot)| - 1$. (The largest label is easily seen to be $2|\mathcal{V}_h(\cdot)| - 1$.) During the traversal, we maintain a list of h "active" generators, i.e., generators associated with the Voronoi cell corresponding to the current vertex in tour. Each edge traversal results in deletion of a generator and insertion of another generator in this list. With each generator, we associate a time interval when it was active during the traversal. Note that each edge traversal ends an interval for a generator (deletion of a generator) and starts a new interval for another generator (insertion of a generator). Therefore, the total number of intervals is $O(|\mathcal{V}_h(\cdot)|)$. We store these intervals in an interval tree $\mathcal{T}_I$. With each Voronoi cell, we also store a time-stamp of the visit of the corresponding vertex of the dual graph during the traversal. (If multiple time-stamps are associated with a vertex, we pick any one.)

To find the generators of the cell containing q, we query $\mathcal{T}_I$ with the time-stamp associated with the cell to find the intervals stabbed by it and report the generators corresponding to the stabbed intervals. Since there are $O(|\mathcal{V}_h(\cdot)|)$ intervals, $\mathcal{T}_I$ uses $O(|\mathcal{V}_h(\cdot)|)$ space and reports the intervals that are stabbed in $O(\log |\mathcal{V}_h(\cdot)| + h) = O(\log n)$ time.

With this technique, the space at any node v of $\mathcal{T}$ (inclusive of the interval tree) is $O(h(|B(v)| - h)) = O(h|B(v)|)$, rather than $O(h^2|B(v)|)$ [61], since generators are not stored explicitly with each cell of $\mathcal{V}_h(v)$. This results in an overall space bound of $O(n \log^2 n)$.

Lemma 3.4. *Let B be a set of blue points in the plane, where $|B| \leq n$. The points of B that are fruitful with respect to a query point q and query distance δ can be computed in $O((1 + k_F) \log n)$ time and $O(n \log^2 n)$ space, where k_F is the number of fruitful points.*

3.3.2.2 Improving the query time

We describe a different way of creating the data structure described in Section 3.3.2 so that the desired fruitful points can be found in $O(\log n + k_F)$ time. The data structure

is built in preprocessing using a probabilistic method; however, the query time is deterministic. The approach follows closely the techniques used by Chazelle et al. [70], so our discussion here is brief. We refer the reader to [70] for details.

One drawback of assigning $h = c \log n$ points in any order to the leaf nodes of $\mathcal{T}$ is that, for some q and δ, it is possible that the number of fruitful points is h (for example) and all of them are stored in a single leaf node. In this case the fruitful points are each assigned a charge of $1/h$ at $O(\log n)$ nodes, which is too much as it leads to the $O((1 + k_F) \log n)$ query time shown in Section 3.3.2.1. To avoid this situation, we use a probabilistic approach to assign points to the leaf nodes of $\mathcal{T}$ is such a way that for any pair of intermediate nodes $v, w \in \mathcal{T}$, if w is a child of v, then the following holds true for any q:

$$|N_{h(w)}(B(w), q) - N_{h(v)}(B(v), q)| \geq d \log |B(w)|,$$

where $N_{h(\cdot)}(B(\cdot), q)$ represents the set of $h(\cdot) = c \log |B(\cdot)|$ points in $B(\cdot)$ that are closest to q and $d > 0$ is a constant. This condition ensures that sufficiently many new fruitful points are "exposed" at w to absorb the cost of exploring at w. It has been shown by Chazelle et al. [70], in the context of circular range search, that there exist constants c, d, and n_0 such that for $n \geq n_0$, the desired assignment is possible with probability at least $1/2$. Their result does not depend on the underlying distance metric and, therefore, is valid for our weighted distance too.

Based on this discussion, we create the data structure mentioned in Section 3.3.2 with the following modifications. We create a complete binary tree $\mathcal{T}$ on points of B and assign $\max\{h, n_0\}$ points of B to the leaves of $\mathcal{T}$ as described above. Also, at each intermediate node $v \in \mathcal{T}$, we build an order-$h(v)$ additively-weighted Voronoi diagram, $\mathcal{V}_{h(v)}(v)$, on $B(v)$ and a point location data structure on $\mathcal{V}_{h(v)}(v)$. The query process remains the same.

It is clear that the space usage for this data structure remains $O(n \log^2 n)$. By an analysis similar to the one in [70] it can be shown that the query time for this approach is $O(\log n + k_F)$.

Lemma 3.5. *Let B be a set of blue points in the plane, where $|B| \leq n$. The points of B that are fruitful with respect to a query point q and query distance δ can be computed in $O(k_F + \log n)$ time using a data structure of size $O(n \log^2 n)$ computed probabilistically, where k_F is the number of fruitful points.*

3.3.3 Putting it all together: The overall algorithm for variable distance 2-CTRQ

Recall that the first step is to find the fruitful blue points with respect to q and δ and the second step is to explore from each fruitful blue point and form and report $(\text{blue}, \text{red})$ tuples. Section 3.3.1, Section 3.3.2.1, and Section 3.3.2.2 have addressed the first problem and the bounds are summarized in Lemma 3.3, Lemma 3.4, and Lemma 3.5, respectively.

The second step is identical to the second step for the fixed distance 2-CTRQ problem in Section 3.2. Theorem 3.1 summarizes the bounds for the fixed distance 2-CTRQ problem and these are also the bounds for the second step for that problem. For simplicity, we will focus on the pair of bounds at the extremes, i.e., $O(n)$ (resp. $O(n \log n)$) space and $O((k+1)\log n)$ (resp. $O(\log n + k)$) query time, corresponding to the parameter value $t = 1$ (resp. $t = \log n$), although it is possible to consider bounds corresponding to other values of t as well.

It is now straightforward to combine a pair of bounds, one for each step, to obtain the overall bounds for the variable distance 2-CTRQ problem. (Note that the number, k_F, of fruitful points is at most the output size k.) Theorem 3.2 below lists the four possibilities. (Of the six possibilities, two have the same space and query time bounds and one is strictly worse than the others; these have been eliminated.)

Theorem 3.2. *A variable distance* 2-CTRQ *problem on a set of n red and blue points can be solved in either (i) $O(\sqrt{n}\log^{O(1)} n + k \log n)$ time using $O(n)$ space, or (ii) $O(\sqrt{n}\log^{O(1)} n + k)$ time using $O(n \log n)$ space, or (iii) $O((1+k)\log n)$ time using $O(n \log^2 n)$ space, or (iv) $O(\log n + k)$ time using $O(n \log^2 n)$ space. (The bounds in (iv) involve a data structure that is built probabilistically in preprocessing; the query time is deterministic.)*

3.4 Handling more than two colors

In this section, we discuss an approach to solve the CTRQ problem for $m > 2$ colors (Problem 3.1) using the solution for 2-CTRQ. Note that $\delta > 0$ can be fixed or variable. As before, we present a two-step solution: In the first step, we find the set of fruitful

points of color c_1 and, in the second step, we explore from each such fruitful point the points of color $c_2, \ldots, c_m$ to form and report the tuples. However, the challenge now is that we have more than two colors and therefore Lemma 3.1 cannot be applied directly to assign weights to the points of color c_1. Also, due to the same reason, in the second step we cannot use circular range search to explore the points of color $c_2, \ldots, c_m$ to form and report the tuples. In the rest of this section we discuss how to overcome these issues.

As mentioned in Section 3.1.1, the idea of fruitful points is central to our approach. As in the case of two colors, we define fruitful points for $m > 2$ colors as follows: For a given ordering, $CS = (c_1, \ldots, c_m)$, a point of color c_1 is *fruitful* if it is part of at least one output tuple for the query point, q, and query distance δ. The following lemma provides a generalization of Lemma 3.1 for $m > 2$ colors.

Lemma 3.6. *For any $p_1 \in C_1$, let $(p_1, \ldots, p_m)$ be an m-tuple of $C_1 \times \cdots \times C_m$ that minimizes $\sum_{i=2}^{m} d(p_{i-1}, p_i)$. Let q be a query point and δ the query distance. Then p_1 is fruitful if and only if $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i) \leq \delta$*

Proof. ($\Leftarrow$) If $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i) \leq \delta$ then the tuple $(p_1, \ldots, p_m)$ is output. Hence, by definition, p_1 is fruitful.

($\Rightarrow$) Suppose p_1 is fruitful. Thus, there is a tuple $(p_1, p'_2, \ldots, p'_m)$ that is output. Thus, $d(q, p_1) + d(p_1, p'_2) + \sum_{i=3}^{m} d(p'_{i-1}, p'_i) \leq \delta$. Since $\sum_{i=2}^{m} d(p_{i-1}, p_i) \leq d(p_1, p'_2) + \sum_{i=3}^{m} d(p'_{i-1}, p'_i)$, we have $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i) \leq \delta$. □

Lemma 3.6 suggests the following preprocessing step to determine the fruitful points at query time. For each $p_1 \in C_1$, we compute an m-tuple, $(p_1, \ldots, p_m)$, of $C_1 \times \cdots \times C_m$ that minimizes $\sum_{i=2}^{m} d(p_{i-1}, p_i)$ and associate with p_1 the real number $\sum_{i=2}^{m} d(p_{i-1}, p_i)$ as a weight $w(p_1)$. The following lemma provides a useful characterization of the desired m-tuple, $(p_1, \ldots, p_m)$.

Lemma 3.7. *For some $p_1 \in C_1$, let $P = (p_1, \ldots, p_m)$ be an m-tuple of $C_1 \times \cdots \times C_m$ that minimizes $\sum_{i=2}^{m} d(p_{i-1}, p_i)$. Then for the point $p_i \in C_i$, $1 \leq i < m$, $P_i = (p_i, \ldots, p_m)$ is an $(m - i + 1)$-tuple of $C_i \times \cdots \times C_m$ that minimizes $\sum_{j=i+1}^{m} d(p_{j-1}, p_j)$.*

Proof. The case $i = 1$ is true by assumption since $P_1 = P$. Let $i > 1$ and assume

that P_i is not optimal for p_i and let $P_i' = (p_i, p_{i+1}', \ldots, p_m') \in C_i \times \cdots \times C_m$ be optimal. Then $\sum_{j=2}^{i} d(p_{j-1}, p_j) + d(p_i, p_{i+1}') + \sum_{j=i+2}^{m} d(p_{j-1}', p_j') < \sum_{j=2}^{i} d(p_{j-1}, p_j) + \sum_{j=i+1}^{m} d(p_{j-1}, p_j) = \sum_{j=2}^{m} d(p_{j-1}, p_j)$, which contradicts the optimality of P. □

Consider the optimal tuple $P = (p_1, \ldots, p_m)$ defined in Lemma 3.7. Then Lemma 3.7 implies that for the point $p_{m-1} \in C_{m-1}$, the point $p_m \in C_m$ minimizes $d(p_{m-1}, p_m)$, i.e., p_m is a nearest neighbor of p_{m-1}. Similarly, for the point $p_{m-2} \in C_{m-2}$, the point $p_{m-1} \in C_{m-1}$ minimizes $d(p_{m-2}, p_{m-1}) + d(p_{m-1}, p_m) = d(p_{m-2}, p_{m-1}) + w(p_{m-1})$, where $w(p_{m-1}) = d(p_{m-1}, p_m)$ is a *weight* assigned to p_{m-1}. In general, for the point $p_i \in C_i$, $1 \leq i < m$, the point $p_{i+1} \in C_{i+1}$ minimizes $d(p_i, p_{i+1}) + w(p_{i+1})$, where $w(p_{i+1}) = \sum_{j=i+2}^{m} d(p_{j-1}, p_j)$ is the weight of p_{i+1}, i.e., $w(p_{i+1})$ is the length of the shortest path from p_{i+1} that visits one point from each of $C_{i+2}, \ldots, C_m$, in that order. (For notational convenience, we take $w(p_m) = 0$.) Thus, p_{i+1} minimizes the sum of $w(p_{i+1})$ and the Euclidean distance between p_i and p_{i+1}.

Our goal is to eventually assign weights (as defined above) to the points of C_1. Based on the above discussion, we preprocess the sets $C_{m-1}, C_{m-2}, \ldots, C_1$, in that order, and assign weights to the points of each set. For any point $p_i \in C_i$, $1 \leq i < m$, $w(p_i)$ is computed by finding the point $p_{i+1} \in C_{i+1}$ that minimizes the sum of $w(p_{i+1})$ and the Euclidean distance between p_i and p_{i+1}. This involves building an additively-weighted Voronoi diagram on the points of C_{i+1}, using the weights found in the previous step (or weight zero if $i + 1 = m$) and querying this with each point of C_i.

With this weight assignment, the problem of finding fruitful points of C_1 for a q and δ boils down to finding (similar to the case of $m = 2$ colors) a set of weighted points $p_1 \in C_1$ such that $d(q, p_1) + w(p_1) \leq \delta$ and, therefore, we can find the fruitful points by using the solution in Section 3.2 or Section 3.3 (for the fixed distance or the variable distance CTRQ problem, respectively).

Recall that the second step is to explore the points of color $c_2, \ldots, c_m$ to form and report the tuples, i.e., for each fruitful point $p_1 \in C_1$, the goal is to identify all the $(m-1)$-tuples $(p_2, \ldots, p_m) \in C_2 \times \ldots \times C_m$ such that $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i) \leq \delta$, i.e., $d(p_1, p_2) + \sum_{i=3}^{m} d(p_{i-1}, p_i) \leq \delta - d(q, p_1)$, i.e., $d(p_1, p_2) + w(p_2) \leq \delta - d(q, p_1)$, i.e., $p_2 \in C_2$ is a fruitful point for the "query" point p_1 and query distance $\delta - d(q, p_1)$ with color sequence $CS' = (c_2, \ldots, c_m)$. (Note that p_2 is an input point that functions as a "query" point here.) Therefore, for the second step, we recursively solve the

variable distance CTRQ problem using the fruitful points as "query" points with query distance and ordering of colors adjusted appropriately. (The problem to be solved is of the variable distance type since successive query distances (e.g., $\delta - d(q, p_1)$) are not known beforehand even if δ itself is.) We terminate the recursion when only two colors are left to process. At this point, we use the solution for variable distance 2-CTRQ. We form tuples as we backtrack and finally report the tuples so formed as the answer set.

3.4.1 Analysis

Let $Q_{fix}(m, n)$ (resp., $Q_{var}(m, n)$) be the query time for the fixed (resp., variable) distance CTRQ problem on n points and m colors. Let $Q_{F,fix}(n)$ (resp., $Q_{F,var}(n)$) be the time to find fruitful points in a set of n points for the fixed (resp., variable) distance CTRQ problem. For $m \geq 3$, the above discussion leads to the following recurrence relations.

$$Q_{fix}(m, n) = Q_{F,fix}(n) + k_{F_2} \times Q_{var}(m - 1, n), \tag{3.1}$$

$$Q_{var}(m, n) = Q_{F,var}(n) + k_{F_2} \times Q_{var}(m - 1, n). \tag{3.2}$$

where k_{F_2} is the number of fruitful points of the second color in CS for q and δ. We use $m = 3$ as the base case for the above recurrences. For $m = 3$, the query time for the fixed distance CTRQ problem is

$$Q_{fix}(3, n) = Q_{F,fix}(n) + k_{F_2} \times Q_{var}(2, n)$$

We use the results mentioned in Theorem 3.2 for $Q_{var}(2, n)$ to derive the corresponding space and time bounds for $Q_{fix}(3, n)$. For example, using $Q_{var}(2, n) = O(\log n + k)$ (with space usage $O(n \log^2 n)$), we get

$$Q_{fix}(3, n) = O(\log n + k_{F_2}) + k_{F_2} O(\log n) + \sum_{i=1}^{k_{F_2}} k_i = O((k + 1) \log n).$$

Here, k_i is the output size for i^{th} fruitful point and $\sum_{i=1}^{k_{F_2}} k_i = k$. (Note that the product of the number of fruitful points of each color is at most the output size k and, hence, the number of fruitful points of each color is at most k.) The space usage for $Q_{F,fix}(n)$ is $O(n)$, therefore, the overall space usage is bounded by the space usage of the variable

distance 2-CTRQ problem, and hence, is $O(n \log^2 n)$. Note that $Q_{fix}(3, n)$ can be further improved to $O(\log n + k)$ by following the technique mentioned in Section 3.2.2 while still using $O(n \log^2 n)$ space. Other space and query time bounds for the fixed distance CTRQ problem, for $m = 3$, are (i) $O(\log n + k\sqrt{n} \log^{O(1)} n)$ time using $O(n)$ space, and (ii) $O((1 + k) \log n)$ time using $O(n \log^2 n)$ space. Similarly, the various space and query time for the variable distance CTRQ problem, for $m = 3$, can be verified to be (i) $O((1 + k)\sqrt{n} \log^{O(1)} n)$ time using $O(n)$ space, (ii) $O((1 + k) \log n)$ time using $O(n \log^2 n)$ space, (iii) $O(\log n + k)$ time using $O(n \log^2 n)$ space (using a probabilistically-computed data structure).

Using $m = 3$ as the base case for the recurrence relations in Equations 3.1 and 3.2, the space and time bounds for the fixed (resp. variable) distance CTRQ problem can be verified to be as listed in Theorem 3.3 (resp. Theorem 3.4). (Of the four possibilities, one is strictly worse than others in both space and time, and is hence eliminated.) Note that the space bounds are independent of m. This is because the structure built on points of color c_i uses space proportional to $|C_i| = n_i$ (e.g. $O(n_i)$ or $O(n_i \log^2 n_i)$), so the total space is proportional to $\sum_{i=1}^{m} n_i = n$.

Theorem 3.3. *A fixed distance* CTRQ *problem on a set of n points in $\mathbb{R}^2$, where each point is assigned a color c_i from a palette of m colors ($m > 2$) can be solved in either (i) $O(\log n + km\sqrt{n} \log^{O(1)} n)$ time using $O(n)$ space, or (ii) $O((1 + km) \log n)$ time using $O(n \log^2 n)$ space, or (iii) $O(\log n + k)$ time using $O(n \log^2 n)$ space. (The bounds in (iii) are based on a probabilistically-computed data structure; the query time is deterministic.)*

Theorem 3.4. *A variable distance* CTRQ *problem on a set of n points in $\mathbb{R}^2$, where each point is assigned a color c_i from a palette of m colors ($m > 2$) can be solved in either (i) $O((1 + km)\sqrt{n} \log^{O(1)} n)$ time using $O(n)$ space, or (ii) $O((1 + km) \log n)$ time using $O(n \log^2 n)$ space, or (iii) $O(\log n + k)$ time using $O(n \log^2 n)$ space. (The bounds in (iii) are based on a probabilistically-computed data structure; the query time is deterministic.)*

Chapter 4

Feature Constraint Colored Tuple Range Query (ConCTRQ)

In this chapter, we extend the CTRQ problem to apply constraints on feature attribute values of the points during tuple identification (Section 1.2.3.1). Specifically, let each point in the plane have some associated feature attributes. The problem we study in this chapter is to report each tuple that satisfies the distance constraint (δ) and such that the points of the tuple satisfy the query constraints on the feature attribute values. We call this the *feature constraint color tuple range query* (ConCTRQ) problem. We show that a certain minimum amount of storage is unavoidable to achieve a fast query time for this problem. Also, solutions with matching upper bounds are given for the fixed distance and variable distance versions of the problem, where δ is known beforehand or is specified as part of the query, respectively.

4.1 Problem formulation

Let S be a set of n points in the plane, which we call the *coordinate space*. Each point $p \in S$ is assigned a color c_i from a palette of m colors ($m \geq 2$). Let $CS = (c_1, \ldots, c_m)$ be a given ordering of the colors. Let C_i be the set of points of color c_i and let $n_i = |C_i|$. Note that $C_i \cap C_j = \emptyset$ if $i \neq j$ and $\bigcup_{i=1}^{m} C_i = S$. Each point $p \in C_i$ has $t_i \geq 1$ additional real-valued attributes, called *features*. We call each $\mathbb{R}^{t_i}$ a *feature space*. Let p' be the point in the feature space ($\mathbb{R}^{t_i}$) corresponding to p and let C_i' be the set of points p'

such that $p \in C_i$. We define the ConCTRQ problem as follows.

Problem 4.1 (ConCTRQ)**.** *Preprocess S into a suitable data structure so that for any query point q in $\mathbb{R}^2$, a distance δ, and a set of axes-parallel query hyper-rectangles $\mathcal{Z} = \{Z_1, \ldots, Z_m\}$ (here Z_i is a t_i-dimensional hyper-rectangle), we can report all tuples $(p_1, \ldots, p_m)$ such that $d(q, p_1) + \sum_{i=2}^{m} d(p_{i-1}, p_i) \leq \delta$ and $p'_i \in C'_i \cap Z_i$ for all i. (Here $d(\cdot, \cdot)$ denotes Euclidean distance.)*

We will first discuss the ConCTRQ problem for $m = 2$ colors and 1-dimensional feature space for points of both colors (i.e., $t_i = 1$, for $i = 1, 2$), which will henceforth be called the $(2, 1)$-ConCTRQ problem. As we will see in Section 4.5, our solution for $(2, 1)$-ConCTRQ generalizes to $m > 2$ colors and $t_i \geq 1$ for all $i \in [1, m]$. (The t_i's need not all be equal.) For simplicity, we re-define the $(2, 1)$-ConCTRQ problem as follows.

Problem 4.2 ($(2, 1)$-ConCTRQ)**.** *Let $B = \{b_1, \ldots, b_{n_1}\}$ be a set of blue points, and $R = \{r_1, \ldots, r_{n_2}\}$ be a set of red points in the plane, $n_1 + n_2 = n$, where each point of $B \cup R$ has an associated real-valued feature attribute. We wish to preprocess the sets, B and R, so that for any query point $q : (x_q, y_q)$ in the plane, a distance δ, and 1-dimensional query ranges $Z_B : [x_B, x'_B]$ and $Z_R : [x_R, x'_R]$, we can report all tuples (b_i, r_j) such that $d(q, b_i) + d(b_i, r_j) \leq \delta$ and $b'_i \in Z_B$, $r'_j \in Z_R$.*

4.2 Contributions

In this chapter, we present the following results for the ConCTRQ problem.

1. In Section 4.3, we establish the computational difficulty of the ConCTRQ problem by showing a polynomial-time reduction from the well-known Set-Intersection problem [71] to the fixed distance $(2, 1)$-ConCTRQ problem. (We define the Set-Intersection problem in Section 4.3.) It is conjectured that, in the cell-probe model of computation, any algorithm for Set-Intersection requires $\tilde{\Omega}((n/\alpha)^2)$ space to achieve a query time of $\tilde{O}(\alpha)$,[1] for $1 \leq \alpha \leq n$. Since the general

[1] The "tilde" notation is commonly used in the discussion of the Set-Intersection problem to suppress polylogarithmic factors; this is for simplicity only.

ConCTRQ problem ($m \geq 2$ and $t_i \geq 1$) is at least as hard as the fixed distance $(2,1)$-ConCTRQ problem, this result immediately implies that the former problem is also hard (in the sense of the Set-Intersection problem).

2. In Section 4.4, we give an efficient solution with matching upper bounds for the fixed distance and variable distance $(2,1)$-ConCTRQ problem. Specifically, our solution for fixed and variable distance $(2,1)$-ConCTRQ problem uses $\tilde{O}(n^2)$ space and has a query time of $\tilde{O}(k)$, where k is the output size. For instance, one possible set of bounds for our solution for a fixed (resp., variable) distance $(2,1)$-ConCTRQ problem uses $O(n^2 \log n)$ (resp., $O(n^2 \log^3 n)$) space and has a query time of $O(\log^3 n + k \log n)$. (Other possible bounds for the fixed and variable distance $(2,1)$-ConCTRQ problem are listed in Theorem 4.1 and Theorem 4.2, respectively.)

3. In Section 4.5, we generalize our solution for $(2,1)$-ConCTRQ problem to the general ConCTRQ problem, i.e. $m > 2$ colors and $t_i \geq 1$ ($i \in [1, m]$). In fact, as we show in Section 4.6, our solution for the $(2,1)$-ConCTRQ problem generalizes to support other types of range constraints in feature space including halfspace range, circular range, simplex range, etc.

4.3 A hardness result

In this section, we establish the computational difficulty of the fixed distance $(2,1)$-ConCTRQ problem by showing a polynomial-time reduction from the so-called Set-Intersection problem, which is conjectured to be difficult [71]. The Set-Intersection problem is defined as follows.

Set-Intersection: We are given a collection of sets $S_1, \ldots, S_m$ of positive reals, where $\sum_{i=1}^{m} |S_i| = n$. We wish to preprocess these sets so that for any query indices u and v, where $u < v$, we can decide if S_u and S_v are disjoint. (The answer is "yes" if and only if S_u and S_v are disjoint.)

It is believed widely that Set-Intersection is "hard" [71]. Specifically, in the so-called cell-probe model without the floor function and where the maximum cardinality of the sets is polylogarithmic in m, any algorithm to answer set intersection queries in

$\tilde{O}(\alpha)$ time requires $\tilde{\Omega}((n/\alpha)^2)$ space, for $1 \leq \alpha \leq n$. In particular, it is unlikely that there exists a solution to this problem that uses low space and simultaneously has low query time (e.g., close-to-linear space and polylogarithmic query time).

We establish our hardness result via the reduction given below. The main idea behind the reduction is as follows: We map the elements of the given sets, $S_1, S_2, \ldots, S_m$, to two colored point-sets, B and R, in the plane. We assign one feature attribute to each of the points of B and R in such a way that S_u and S_v are disjoint if and only if the answer to a fixed distance $(2,1)$-ConCTRQ problem on B and R for some q, δ, and 1-dimensional ranges, Z_B and Z_R, on the feature space of B and R, is the empty set. (Z_B and Z_R depend on S_u and S_v, respectively.)

The reduction Consider an instance of Set-Intersection. For some $\varepsilon > 0$, let C_B and C_R be two circles in the plane with radius 1 and $1+\varepsilon$, respectively. We map each element of each set S_t, for $1 \leq t \leq m$, to two points, one on C_B and another one on C_R and assign one feature attribute to each of the new points as follows: Let $n_0 = 0$ and $n_t = n_{t-1} + |S_t|$, for $1 \leq t \leq m$. Let e_{tk} be the k^{th} element of S_t. (S_t is unordered and e_{tk} is simply the k^{th} element in an arbitrary ordering of the elements of S_t.) Define the line $L : y = e_{tk}x$. We create a blue (resp., red) point at the intersection of L and C_B (resp., L and C_R) in the first quadrant and assign it a feature attribute value of $k + n_{t-1}$. (See Figure 4.1.) Let O be the set of newly created points. This preprocessing takes $O(n)$ time and results in a set of $2n$ colored points.

We can answer a Set-Intersection query as follows: We create the data structure to solve the fixed distance $(2,1)$-ConCTRQ problem on O with $\delta = 1+\varepsilon$. Given query indices u and v, where $u < v$, we query the fixed distance $(2,1)$-ConCTRQ structure with $q = (0,0)$, $Z_B = [n_{u-1}+1,\, n_u]$, and $Z_R = [n_{v-1}+1,\, n_v]$. If the query outputs at least one (blue, red) pair, then we output "no" as the answer to Set-Intersection. Otherwise, we output "yes".

Lemma 4.1. *The answer to* Set-Intersection*, for query indices u and v ($u < v$), is "no" if and only if the fixed distance $(2,1)$-ConCTRQ problem with $q = (0,0)$, $\delta = 1+\varepsilon$, $Z_B = [n_{u-1}+1,\, n_u]$, and $Z_R = [n_{v-1}+1,\, n_v]$, outputs at least one (blue, red) tuple.*

Proof. ($\Rightarrow$) Suppose that the answer to Set-Intersection, for query indices u and v

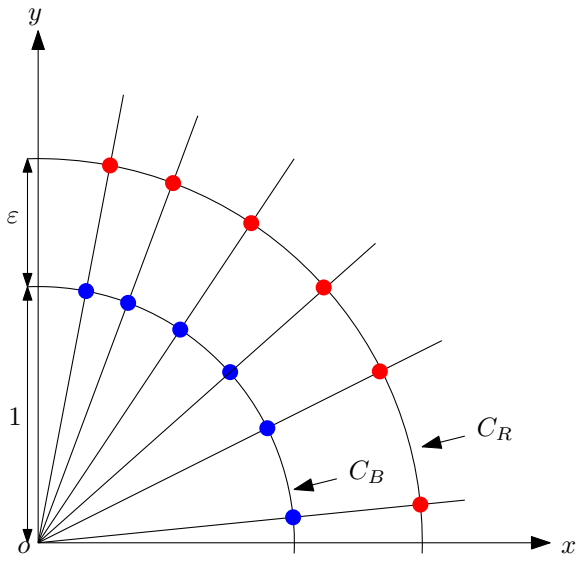
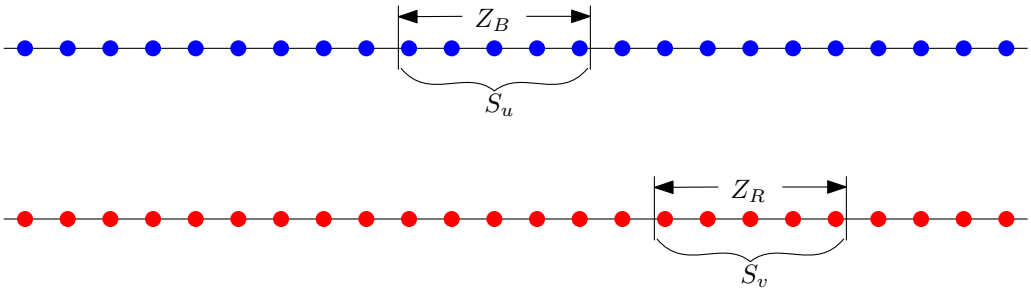


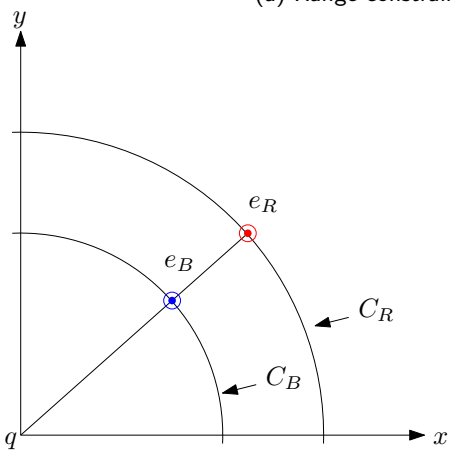
Figure 4.1: Example of the reduction. Each element e is shown as a line $y = ex$, with a blue point and a red point at the intersection with circles C_B and C_R, respectively.

$(u < v)$, is "no". Thus, there is an element, e, in $S_u \cap S_v$. Let e_B (resp., e_R) be the blue (resp., red) point on C_B (resp., C_R) to which e is mapped. Hence, e_B and e_R both lie on the line $y = ex$. Since e_B and e_R are collinear, $d(q, e_B)+d(e_B, e_R) = d(q, e_R) = 1+\varepsilon = \delta$. (See Figure 4.2b.) Moreover, it is easy to verify that for any $e \in S_u$ (resp., $e \in S_v$), $e'_B \in Z_B$ (resp., $e'_R \in Z_R$). (See Figure 4.2a.) Thus, the fixed distance $(2, 1)$-ConCTRQ problem with $\delta = 1 + \varepsilon$, for the query tuple (q, Z_B, Z_R), outputs the (blue, red) pair (e_B, e_R).

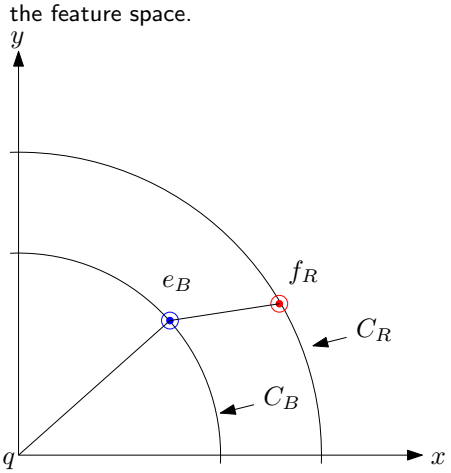
($\Leftarrow$) Suppose that the answer to Set-Intersection, for query indices u and v $(u < v)$, is "yes". Thus, $S_u \cap S_v = \emptyset$. Let e_B and f_R be points in B and R, respectively, corresponding to any elements $e \in S_u$ and $f \in S_v$. We know that $e \neq f$, since $S_u \cap S_v = \emptyset$. Hence, e_B and f_R are not collinear. We have $d(q, e_B) + d(e_B, f_R) > 1 + \varepsilon = \delta$, by the triangle inequality. (See Figure 4.2c.) Thus, the $(2, 1)$-ConCTRQ problem with $\delta = 1 + \varepsilon$, for the query tuple (q, Z_B, Z_R), outputs no (blue, red) tuple. □



(a) Range constraints in the feature space.



(b) e_B and e_R are collinear. Therefore, $d(q, e_B) + d(e_B, e_R) = 1 + \varepsilon = \delta$.



(c) e_B and f_R are not collinear. Therefore, $d(q, e_B) + d(e_B, f_R) > 1 + \varepsilon = \delta$.

Figure 4.2: Illustrating the correctness of the reduction. (The figure is meant to be schematic and is not drawn to scale.)

4.4 $(2, 1)$-ConCTRQ

In this section, we present our solution for the fixed and variable distance versions of the $(2, 1)$-ConCTRQ problem (Problem 4.2). For the sake of simplicity, we first present the solution for the fixed distance version of the problem. We discuss the changes required for the variable distance version of the problem at the end of this section.

In preprocessing, we build a 1-dimensional range tree, T_B, [2] on the feature attribute values of the points in B. At each node $v \in T_B$, we store another 1-dimensional range, $T_R(v)$, built on the feature attribute values of all the points in R. For any node $v \in T_B$, let $B(v)$ be the set of points of B stored in the leaves of v's subtree. Also, for any node

$u \in T_R(v)$, let $R(u)$ be the set of points of R stored in the leaves of u's subtree. At u we store the data structure, $DS_{CTRQ}(u)$, for the fixed distance 2-CTRQ problem [10], built on $B(v)$ and $R(u)$ (Figure 4.3). This yields a multi-tree structure consisting of a range tree on B, each of whose nodes points to a range tree on R. In turn, each node of the range tree on R points to a fixed distance 2-CTRQ structure (Section 3.2).

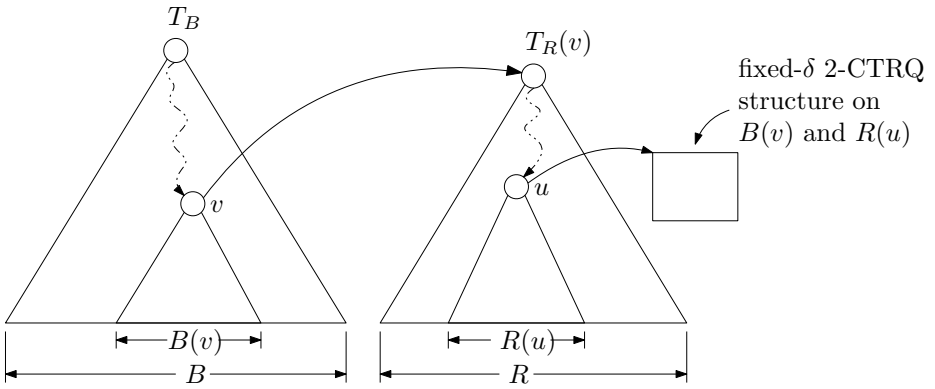


Figure 4.3: An example of the multi-tree structure built for $(2, 1)$-ConCTRQ problem.

Given q and feature constraints $Z_B = [x_B, x'_B]$ and $Z_R = [x_R, x'_R]$, we first query T_B with range Z_B to find a set $\mathcal{V}_B$ of $O(\log n)$ nodes of T_B that covers the range Z_B; each such node is called *canonical node* [2]. Next, at each $v \in \mathcal{V}_B$, we query $T_R(v)$ with range Z_R to find a set $\mathcal{V}_R(v)$ of $O(\log n)$ canonical nodes that covers the range Z_R. Let $\mathcal{V} = \cup_{v \in \mathcal{V}_B} \mathcal{V}_R(v)$; clearly $|\mathcal{V}| = O(\log^2 n)$. For each node $u \in \mathcal{V}$, we query the associated fixed distance 2-CTRQ data structure, $DS_{CTRQ}(u)$, with q and output the tuples reported as the answer for the fixed distance $(2, 1)$-ConCTRQ problem.

We argue that a tuple (b_i, r_j) is reported if and only if $b'_i \in Z_B$, $r'_j \in Z_R$, and $d(q, b_i) + d(b_i, r_j) \leq \delta$. Suppose that (b_i, r_j) is reported. Thus, (b_i, r_j) is found when $DS_{CTRQ}(u)$ is queried for some $u \in \mathcal{V}$, i.e., $d(q, b_i) + d(b_i, r_j) \leq \delta$. Also, $u \in \mathcal{V}_R(v)$ for some $v \in \mathcal{V}_B$, therefore $b'_i \in B(v)$ and $r'_j \in R(u)$, i.e., $b'_i \in Z_B$ and $r'_j \in Z_R$. On the other hand, suppose that $d(q, b_i) + d(b_i, r_j) \leq \delta$ for some b_i and r_j such that $b'_i \in Z_B$ and $r'_j \in Z_R$. Then, there exists a $u \in T_R(v)$ for some $v \in T_B$ such that $b'_i \in B(v)$ and $r'_j \in R(u)$. Also, $v \in \mathcal{V}_B$ and $u \in \mathcal{V}_R(v)$, therefore, $DS_{CTRQ}(u)$ is queried, and hence (b_i, r_j) is reported. This establishes the correctness of the query algorithm.

Analysis. Let $Q(n) = O(f(n) + k \cdot g(n))$ and $S(n)$ be the query time and space required, respectively, for the data structure to solve the fixed distance 2-CTRQ problem on n blue and red points. As mentioned before $|\mathcal{V}| = O(\log^2 n)$, i.e., the number of nodes queried for 2-CTRQ is $O(\log^2 n)$, therefore the total query time is $\sum_{i=1}^{\log^2 n} O(f(n) + k_i \cdot g(n)) = O(f(n) \cdot \log^2 n + k \cdot g(n))$, where k_i is the number of tuples reported at the i^{th} node and $\sum_{i=1}^{\log^2 n} k_i = k$ is the total number of tuples reported. The total space required by the $DS_{CTRQ}(\cdot)$ structures stored at the nodes of any height h of $T_R(v)$, for some $v \in T_B$, is $O(S(n))$ since $S(n)$ is at least linear. Therefore the space required for $T_R(v)$ is $O(S(n) \cdot \log n)$. Since the number of nodes in T_B is $O(n)$, the total space required by T_B is $O(S(n) \cdot n \log n)$.

By using the bounds for the fixed distance 2-CTRQ problem listed in Theorem 3.1, we get the overall bounds for the fixed distance $(2, 1)$-ConCTRQ problem as follows.

Theorem 4.1. *A fixed distance* $(2, 1)$-ConCTRQ *problem on a set of* n *red and blue points can be solved in* $O(\log^3 n + k \log n)$ *(resp.,* $O(\log^3 n + k)$*) time using* $O(n^2 \log n)$ *(resp.,* $O(n^2 \log^2 n)$*) space.*

Handling variable distance queries. The query range constraints Z_B and Z_R do not depend on the query distance being fixed. Therefore, as before, we create the multi-tree structure on the feature attribute values of points in B and R to handle the range constraints on the feature space. To handle the variable distance constraint, we replace the fixed distance 2-CTRQ structure associated with the nodes of $T_R(v)$, for all $v \in T_B$, with the variable distance 2-CTRQ structure.

By following an analysis similar to the one for the fixed distance version of the problem along with the bounds for the variable distance 2-CTRQ problem listed in Theorem 3.2, we get the overall bounds for the variable distance $(2, 1)$-ConCTRQ problem as follows.

Theorem 4.2. *A variable distance* $(2, 1)$-ConCTRQ *problem on a set of* n *red and blue points can be solved in either* (i) $O(\sqrt{n} \log^{O(1)} n + k \log n)$ *time using* $O(n^2 \log n)$ *space, or* (ii) $O(\sqrt{n} \log^{O(1)} n + k)$ *time using* $O(n^2 \log^2 n)$ *space, or* (iii) $O(\log^3 n + k \log n)$ *time using* $O(n^2 \log^3 n)$ *space, or* (iv) $O(\log^3 n + k)$ *time using* $O(n^2 \log^3 n)$ *space. (The bounds in* (iv) *involve a data structure that is built probabilistically in preprocessing; the query time is deterministic.)*

4.5 Handling more than two colors

In this section, we generalize our approach for the $(2,1)$-ConCTRQ problem to solve the ConCTRQ problem (Problem 4.1) for $m > 2$ colors and $t_i \geq 1$ for all $i \in [1, m]$. Note that $\delta > 0$ can be fixed or variable.

Similar to the $(2,1)$-ConCTRQ problem, for the ConCTRQ problem, we build a multi-tree structure augmented with the data structure for the CTRQ problem (Section 3.4, [10]). In preprocessing, we build a multi-tree structure on the feature attribute values of the points of S, as follows. We first build a t_1-dimensional range tree, T_{C_1}, on the feature attribute values of points in C_1. For $i = 2, \ldots, m$, we augment each node $v_{i-1} \in T_{C_{i-1}}$ at level t_{i-1} with a t_i-dimensional range tree, $T_{C_i}(v_{i-1})$, built on the feature attribute values of all points in C_i.

For any node v_m at the level t_m of T_{C_m}, consider the path from the root of T_{C_1} to v_m. For each $i \in [1, m]$, let v_i be the last level-t_i node on the path. Also, let $C_1(v_1), C_2(v_2), \ldots, C_m(v_m)$ be the points of $C_1, C_2, \ldots, C_m$ stored in the subtree rooted at the nodes $v_1, v_2, \ldots v_m$, respectively. At v_m, we store the data structure, $DS_{CTRQ}(v_m)$, for fixed (resp., variable) distance CTRQ problem (Section 3.4), built on the points of $C_1(v_1), C_2(v_2), \ldots, C_m(v_m)$, for the fixed (resp., variable) distance ConCTRQ problem. For example, Figure 4.4 shows the tree structure for ConCTRQ problem for $m = 3$ sets of colored points. Here, the points of C_1 and C_3 have two feature attributes (i.e., $t_1 = t_2 = 2$), and the points of C_2 have one feature attribute (i.e., $t_2 = 1$).

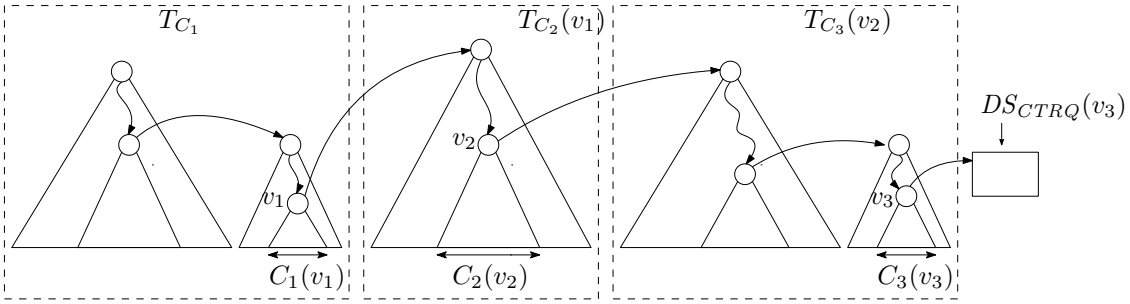


Figure 4.4: An example of the multi-tree structure built for ConCTRQ problem with $m = 3$. Here the points of C_1 and C_3 have two feature attributes and the points of C_2 have one feature attribute.

Given q (or q, δ for variable distance ConCTRQ), and a set of axes-parallel hyper-rectangles $\mathcal{Z} = \{Z_1, \ldots, Z_m\}$, we first find a set of canonical nodes, $\mathcal{V}_1$, at level t_1 of T_{C_1} that covers the range Z_1. For $i = 2, \ldots, m$, we search with Z_i in $T_{C_i}(v)$, for all $v \in \mathcal{V}_{i-1}$, and find a set of canonical nodes $\mathcal{V}_i$ at level t_i of T_{C_i}. Clearly, $|\mathcal{V}_m| = O(\log^{d_m} n)$, where $d_m = \sum_{i=1}^{m} t_i$. Finally, for each node $v \in \mathcal{V}_m$, we query the associated fixed (resp., variable) distance CTRQ data structure, $DS_{CTRQ}(v)$, with q (resp., q and δ) and output the tuples reported as answer for the fixed (resp., variable) distance ConCTRQ problem.

By an analysis similar to the one for $(2,1)$-ConCTRQ (Section 4.4) along with the bounds for the CTRQ problem (Theorem 3.3 and Theorem 3.4), we get following bounds for the (fixed distance or variable distance) ConCTRQ problem.

Theorem 4.3. *A fixed distance* ConCTRQ *problem on a set of n points in the plane, where each point is assigned a color c_i from a palette of m colors ($m \geq 2$) and has t_i feature attributes, can be solved in either (i) $O(\log^{d_m+1} n + km\sqrt{n}\log^{O(1)} n)$ time using $O(n^m \log^{d_m} n)$-space data structure, or (ii) $O(\log^{d_m+1} n+k)$ time using $O(n^m \log^{d_m+2} n)$-space probabilistically-computed data structure. Here, $d_m = \sum_{i=1}^{m} t_i$.*

Theorem 4.4. *A variable distance* ConCTRQ *problem on a set of n points in the plane, where each point is assigned a color c_i from a palette of m colors ($m \geq 2$) and has t_i feature attributes, can be solved in either (i) $O((\log^{d_m} n + km)\sqrt{n}\log^{O(1)} n)$ time using $O(n^m \log^{d_m} n)$-space data structure, or (ii) $O(\log^{d_m+1} n + k)$ time using $O(n^m \log^{d_m+2} n)$-space probabilistically-computed data structure. Here, $d_m = \sum_{i=1}^{m} t_i$.*

4.6 Handling other query range types

Our approach for ConCTRQ can be easily modified to handle various other types of query ranges including halfspace range, circular range, simplex range, etc. by replacing the range tree with an appropriate tree structure (e.g., a partition tree). For example, consider the ConCTRQ problem with $m = 3$ colors, where the feature constraints for color c_1 and c_3 are orthogonal hyper-rectangles (like the regular ConCTRQ problem) but the feature constraints for color c_2 are halfspace ranges.

To solve this instance of the ConCTRQ problem, we create a multi-tree structure of Section 4.5 with the range tree for the points of color c_2 replaced with the corresponding

partition tree [2]. Specifically, we first build a range tree, T_{C_1} on the feature attribute values of the points of color c_1. At each node v_1 on the last level of T_{C_1}, we build a partition tree $T_{C_2}(v_1)$ on the feature attribute values of the points of color c_2 and at each internal node v_2 of $T_{C_2}(v_1)$, we build a range tree, $T_{C_3}(v_2)$, on the feature attribute values of the points of color c_3. Finally at each node v_3 on the last level of $T_{C_3}(v_2)$, we build a CTRQ structure on the appropriate subset of points of color c_1, c_2, and c_3. (We omit the discussion of the analysis as it mirrors the analysis of the ConCTRQ problem with orthogonal feature constraints.)

Chapter 5

Threshold Constraint Colored Tuple Range Query (τ-CTRQ)

In this chapter, we consider colored points where each point has an associated real-valued score. The score of a point can be computed from the feature attribute values beforehand. For any query point q, a distance constraint δ, and a query threshold value τ, we wish to report each point-tuple such that the distance traveled from q to visit the points of the tuple is at most δ and the score of the tuple is at least τ. Here the score of a tuple is defined as the sum of scores of the points in the tuple. We call this the *threshold constraint color tuple range query* (τ-CTRQ) problem. We give efficient solutions for the four versions of the τ-CTRQ problem based on whether δ and τ are known during preprocessing or specified at the query time.

5.1 Problem formulation

Let S be a set of n points in the plane. Each point $p \in S$ is assigned a color c_i from a palette of m colors ($m \geq 2$) and has an associated score $s(p)$ ($0 \leq s(p) \leq 1$). Let $CS = (c_1, \ldots, c_m)$ be a given ordering of the colors. Let C_i be the set of points of color c_i and let $n_i = |C_i|$. Note that $C_i \cap C_j = \emptyset$ if $i \neq j$ and $\bigcup_{i=1}^{m} C_i = S$.

Throughout, we will consider m-tuples $t = (p_1, \ldots, p_m) \in S^m$. The *score* of t is $s(t) = \sum_{i=1}^{m} s(p_i)$. The *path length* of t, denoted by $l_{path}(t)$, is the distance traveled from p_1 to p_m, in the order specified by t, i.e., $l_{path}(t) = \sum_{i=1}^{m-1} d(p_i, p_{i+1})$, where $d(\cdot, \cdot)$

denote Euclidean distance.

With these definitions in hand, we can define the *threshold constraint color tuple range query* (τ-CTRQ) problem that we wish to solve, as follows.

Problem 5.1 (τ-CTRQ)**.** *Preprocess S into a suitable data structure so that for any query point q, a distance $\delta > 0$, and a threshold $\tau \geq 0$, we can report efficiently all tuples $t = (p_1, \ldots, p_m)$ such that $d(q, p_1) + l_{path}(t) \leq \delta$ and $s(t) \geq \tau$.*

For ease of exposition, we will first discuss the τ-CTRQ problem for $m = 2$ colors, which will, henceforth, be called the $(\tau, 2)$-CTRQ problem. As we will discuss in Section 5.6, our solution for $(\tau, 2)$-CTRQ generalizes to $m > 2$ colors. For simplicity, we re-define the $(\tau, 2)$-CTRQ problem as follows. (Path length of tuple $t = (b_i, r_j)$ is simply the distance between b_i and r_j, i.e. $l_{path}(t) = d(b_i, r_j)$ and score of t is $s(t) = s(b_i) + s(r_j)$.)

Problem 5.2 ($(\tau, 2)$-CTRQ)**.** *Let $B = \{b_1, \ldots, b_{n_1}\}$ be a set of blue points, and $R = \{r_1, \ldots, r_{n_2}\}$ be a set of red points in the plane, where $n_1 + n_2 = n$ and each point $p \in B \cup R$ has an associated score $s(p)$ $(0 \leq s(p) \leq 1)$. We wish to preprocess the sets, B and R, so that for any query point $q = (x_q, y_q)$ in the plane, a distance $\delta > 0$, and a threshold $\tau \geq 0$, we can report all tuples $t = (b_i, r_j)$ such that $d(q, b_i) + l_{path}(t) = d(q, b_i) + d(b_i, r_j) \leq \delta$ and $s(t) = s(b_i) + s(r_j) \geq \tau$.*

Depending upon the application, $\delta > 0$ and $\tau \geq 0$ can both be "fixed" (known during preprocessing) or "variable" (specified only at query time). Therefore, we have four versions of the problem that are of interest, which we call *fixed-distance fixed-threshold*, *fixed-distance variable-threshold*, *variable-distance fixed-threshold*, and *variable-distance variable-threshold versions* of $(\tau, 2)$-CTRQ. We first present our solution for the two fixed-distance versions of the problem in Section 5.3 and Section 5.4 and then extend these to the corresponding variable-distance counterparts in Section 5.5. We extend these solutions to $m > 2$ colors, i.e., to τ-CTRQ, in Section 5.6.

5.2 Contributions

Similar to the solution for the CTRQ problem presented in Chapter 3, the algorithms here are based on a two-step approach. Specifically, in the first step we identify an

appropriate subset of the points of the first color of the given color sequence that are guaranteed to be part of at least one reported tuple for a query triple (q, δ, τ). We call such points *fruitful* points. In the second step, we explore from each fruitful point to identify the points of the remaining colors (in the order specified by CS) that satisfy the query, and output the corresponding tuples. Table 5.1 and Table 5.2 summarizes the space and query times of our algorithms for the $(\tau, 2)$-CTRQ and the τ-CTRQ problem, respectively.

Table 5.1: Summary of results for the $(\tau, 2)$-CTRQ problem. Here, α is a user specified parameter, $\alpha \in [\log n, \log^2 n]$. The query time marked with "$*$" is with probabilistic preprocessing (the query time is deterministic). The space bounds marked with "†" are expected.

	Fixed-δ		Variable-δ	
	$Q(n)$	$S(n)$	$Q(n)$	$S(n)$
Fixed-τ	$O(\log n + \frac{k}{\alpha}\log^2 n)$	$O(n\alpha)$	$O(\sqrt{n}\log^{O(1)} n + \frac{k}{\alpha}\log^2 n)$ $O((k+1)\log n)$ $O(\log n + k)^*$	$O(n\alpha)$ $O(n\log^2 n)$ $O(n\log^2 n)$
Variable-τ	$O((k+1)\log n)$	$O(n\log n)^\dagger$	$O(\sqrt{n}\log^{O(1)} n + k\log^2 n)$ $O((k+1)\log^2 n)$	$O(n\log^2 n)^\dagger$ $O(n\log^3 n)^\dagger$

Table 5.2: Summary of results for the τ-CTRQ problem for $m > 2$ colors. The space bounds are expected.

	Fixed-δ		Variable-δ	
	$Q(n)$	$S(n)$	$Q(n)$	$S(n)$
Fixed-τ	$O(\log n + km\sqrt{n}\log^{O(1)} n)$ $O(\log n + km\log^2 n)$	$O(n\log^2 n)$ $O(n\log^3 n)$	$O(\log n + km\sqrt{n}\log^{O(1)} n)$ $O(\log n + km\log^2 n)$	$O(n\log^2 n)$ $O(n\log^3 n)$
Variable-τ	$O(\log^2 n + km\sqrt{n}\log^{O(1)} n)$ $O((1+km)\log^2 n)$	$O(n\log^2 n)$ $O(n\log^3 n)$	$O((1+km)\sqrt{n}\log^{O(1)} n)$ $O((1+km)\log^2 n)$	$O(n\log^2 n)$ $O(n\log^3 n)$

5.3 Fixed-distance fixed-threshold $(\tau, 2)$-CTRQ

In this section, we present our two-step solution for the fixed-distance and fixed-threshold version of the $(\tau, 2)$-CTRQ problem (Problem 5.2 with δ and τ prespecified).

5.3.1 Step 1: Identify fruitful blue points

Recall that a blue point is fruitful if and only if it is part of at least one reported (blue, red) pair for a given q, δ, and τ. The following lemma provides a useful characterization of such fruitful blue points.

Lemma 5.1. *Let q be a query point, $\delta > 0$ the (fixed) query distance, and $\tau \geq 0$ the (fixed) query threshold. For any $b_i \in B$, let $r_j \in R$ be its closest red point such that $s(r_j) \geq \tau - s(b_i)$. Then b_i is fruitful if and only if $d(q, b_i) + d(b_i, r_j) \leq \delta$.*

Proof. ($\Leftarrow$) Suppose $d(q, b_i) + d(b_i, r_j) \leq \delta$. Since $s(r_j) \geq \tau - s(b_i)$, i.e., $s(b_i) + s(r_j) \geq \tau$, the tuple ($b_i$, r_j) is output. Hence, by definition, b_i is fruitful.

($\Rightarrow$) Suppose b_i is fruitful. Therefore by definition, there is an $r_k \in R$ such that (b_i, r_k) is output, i.e., $d(q, b_i) + d(b_i, r_k) \leq \delta$ and $s(b_i) + s(r_k) \geq \tau$ or $s(r_k) \geq \tau - s(b_i)$. Since, among the red points with score at least $\tau - s(b_i)$, r_j is the nearest red point to b_i, we have $d(b_i, r_j) \leq d(b_i, r_k)$, therefore $d(q, b_i) + d(b_i, r_j) \leq \delta$. □

Based on Lemma 5.1, we propose the following preprocessing steps for a given q, δ, and τ to help identify the fruitful blue points at query time. For each $b_i \in B$ with score $s(b_i)$, we compute its nearest red point $r_j \in R$ with score $s(r_j) \geq \tau - s(b_i)$. We associate with b_i the distance $d(b_i, r_j)$ as a *weight* $w(b_i)$. Now, for a query point q, b_i is fruitful if and only if $d(q, b_i) + w(b_i) \leq \delta$, i.e., $d(q, b_i) \leq \delta - w(b_i)$. We build the fruitful point-finding data structure for the fixed distance 2-CTRQ problem (Section 3.2) on this weighted set of blue points. Specifically, for each blue point b_i with weight $w(b_i)$, we create a disk D_i of radius of radius $\delta - w(b_i)$ centered at b_i. Note that, the disk D_i contains or is "stabbed" by q if and only if $d(q, b_i) \leq \delta - w(b_i)$, i.e., b_i is fruitful for q and δ, by Lemma 5.1. We build a disk stabbing data structure DS_f on the disks D_i that can identify the disks stabbed by q, and hence can identify the fruitful blue points.

As shown in [34], the disk stabbing problem in $\mathbb{R}^2$ can be transformed to a halfspace range search query problem in $\mathbb{R}^3$ using a lifting map followed by geometric duality. Afshani et al. [35] showed that the halfspace range search query problem in $\mathbb{R}^3$ on n points can be answered in $O(\log n + k)$ time using an $O(n)$ space data structure, where k is the output size. This leads to the following result for the first step of the algorithm.

Lemma 5.2. *Let B be a set of blue point and R be a set of red points, where $|B|+|R| = n$. There exists a data structure of size $O(n)$ which can report the k_F points of B that are fruitful with respect to q, (fixed) δ, and (fixed) τ in $O(\log n + k_F)$ time.*

5.3.2 Step 2: Explore red points and output tuples

Recall that the second step of the query algorithm is to explore the red points for each fruitful blue point to form and report the valid (blue, red) tuples. Note that, for a fruitful blue point b_i with score $s(b_i)$, only the red points that are reachable within distance $\delta - d(q, b_i)$ from b_i and have score at least $\tau - s(b_i)$ can form valid tuples. A naïve way to find such red points is to first find all the red points that are reachable within distance $\delta - d(q, b_i)$ from b_i (via a circular range search) and then filter out the red points with score less than $\tau - s(b_i)$. Unfortunately, this could be very expensive as the circular range search may find a lot of red points but very few of them satisfy the score constraint. Therefore, we propose the following preprocessing to build a data structure that solves this problem efficiently.

We build a 1-dimensional range tree [2], T_R, on the distinct values of the scores of the red points, where the red points with same scores are stored at a single leaf node. At any node $v \in T_R$, let $R(v)$ denote the set of points stored at the subtree rooted at v (or at v, if v is a leaf node). At v, we build a circular range search data structure, $DS_c(v)$, on $R(v)$.

To explore red points from a fruitful blue point b_i and output tuples for a given q, δ, and τ, we query T_R with range $Z_i = [\tau - s(b_i), \infty)$, and find a set $\mathcal{V}_R(b_i)$ of $O(\log n)$ *canonical* nodes of T_R that cover the range Z_i [2]. Clearly, for each $v \in \mathcal{V}_R(b_i)$, the scores of the red points stored in the subtree rooted at v is at least $\tau - s(b_i)$. For each $v \in \mathcal{V}_R(b_i)$, we query the associated $DS_c(v)$ with a disk of radius $\delta - d(q, b_i)$ centered at b_i to find the desired set of red points and report the tuples.

Similar to the disk stabbing problem, the circular search query in $\mathbb{R}^2$ can be transformed to a halfspace range search query problem in $\mathbb{R}^3$ using a lifting map [34]. Therefore, for any $v \in T_R$, $DS_c(v)$ can be implemented in $O(|R(v)|)$ space by using the solution for the halfspace range search query problem in $\mathbb{R}^3$ [35]. Hence, the overall space required for T_R is $\sum_{v \in T_R} O(|R(v)|) = O(n \log n)$.

For a fruitful point b_i, the query on T_R to find the set $\mathcal{V}_R(b_i)$ of $O(\log n)$ canonical

nodes takes $O(\log n)$ time. For each canonical node $v \in \mathcal{V}_R(b_i)$, the query on $DS_c(v)$ to report tuples takes $O(\log n + k_i(v))$, where $k_i(v)$ is the number of the number of red points in the subtree rooted at v that form valid tuples with b_i. Therefore, for each fruitful point, the total runtime for the second step is $O(\log n)+\sum_{v\in\mathcal{V}_R(b_i)} O(\log n+k_i) = O(\log^2 n + k_i)$, where $k_i = \sum_{v\in\mathcal{V}_R(b_i)} k_i(v)$ is the total number of tuples of the answer set that contain b_i.

Lemma 5.3. *Let B be a set of blue points and R be a set of red points, where $|B|+|R| = n$. There exists a data structure of size $O(n \log n)$ and query time $O(\log^2 n+k_i)$ to report the k_i points of R that form tuples with a fruitful point b_i for a given q, δ, and τ.*

5.3.3 Overall algorithm

It is now straightforward to combine the two steps of the algorithm to get the overall solution for the fixed-distance fixed-threshold $(\tau, 2)$-CTRQ problem. The overall preprocessing and query algorithm is shown as Algorithm 6 and 7, respectively.

We argue that a tuple (b_i, r_j) is reported if and only if $d(q, b_i) + d(b_i, r_j) \leq \delta$ and $s(b_i) + s(r_j) \geq \tau$. Suppose that (b_i, r_j) is reported. Thus, r_j is found when $DS_t(v)$ is queried, for some $v \in \mathcal{V}_R(b_i)$, with a disk of radius $\delta - d(q, b_i)$ centered at b_i, i.e., $d(b_i, r_j) \leq \delta - d(q, b_i)$, i.e., $d(q, b_i) + d(b_i, r_j) \leq \delta$. Also, for any $r_k \in \cup_{v\in\mathcal{V}_R(b_i)} R(v)$, $s(r_k) \geq \tau - s(b_i)$, therefore $s(r_j) \geq \tau - s(b_i)$, i.e., $s(b_i) + s(r_j) \geq \tau$. On the other hand, suppose that $d(q, b_i) + d(b_i, r_j) \leq \delta$ and $s(b_i) + s(r_j) \geq \tau$ for some tuple (b_i, r_j). Then, $d(q, b_i) + d(b_i, nn_i^\tau) \leq \delta$ where nn_i^τ is the nearest red point to b_i with $s(nn_i^\tau) \geq \tau - s(b_i)$, i.e., $s(b_i) + s(nn_i^\tau) \geq \tau$. Thus, by Lemma 5.1, b_i is a fruitful point. Hence T_R is queried with range $Z_i = [\tau - s(b_i), \infty)$ to find the set, $\mathcal{V}_R(b_i)$, of canonical nodes that cover Z_i. Since $s(b_i)+s(r_j) \geq \tau$, i.e., $s(r_j) \geq \tau - s(b_i)$, it follows that $r_j \in R(v)$ for some $v \in \mathcal{V}_R(b_i)$. Thus $DS_t(v)$ is queried with b_i and radius $\rho_i = \delta - d(q, b_i) \geq \delta - (\delta - d(b_i, r_j)) = d(b_i, r_j)$, and r_j is found by the query. Hence, (b_i, r_j) is reported. This establishes the correctness of the algorithm.

Analysis: It is easy to combine the bounds the two steps to get the overall query time and space requirement for the fixed-distance fixed-threshold $(\tau, 2)$-CTRQ problem. The first step takes $O(\log n + k_F)$ time, where $k_F = |F|$ is the number of fruitful points (Lemma 5.2). For each fruitful point $b_i \in F$, the second step takes $O(\log^2 n + k_i)$

Algorithm 6 : Preprocessing

Input: B, R: sets of input points; δ, τ: reals.
Output: DS_f: Data structure to find fruitful points; T_R: Data structure to form tuples.

1: $D \leftarrow \emptyset$ // `set of disks`
2: **for all** $b_i \in B$ **do**
3: $nn_i^\tau \leftarrow$ nearest red point of b_i with score at least $\tau - s(b_i)$.
4: **if** $d(b_i, nn_i^\tau) \leq \delta$ **then** // b_i `cannot be fruitful if` $d(b_i, nn_i^\tau) > \delta$
5: $w(b_i) \leftarrow d(b_i, nn_i^\tau)$
6: $D_i \leftarrow$ disk with radius $\delta - w(b_i)$ and centered at b_i
7: $D \leftarrow D \cup \{D_i\}$
8: **end if**
9: **end for**
10: $DS_f \leftarrow$ Create disk stabbing data structure on disks in D.
11: $T_R \leftarrow$ 1-dimensional range tree on distinct values of the scores of the red points.
12: **for all** leaf nodes v of T_R **do**
13: $s(v) \leftarrow$ score associated with the leaf node v.
14: $R(v) \leftarrow \{r_j \in R | s(r_j) = s(v)\}$.
15: $DS_t(v) \leftarrow$ create circular range search data structure on the points in $R(v)$.
16: **end for**
17: **for all** non-leaf nodes v of T_R **do**
18: $R(v) \leftarrow$ set of points (of R) stored in the subtree rooted at v.
19: $DS_t(v) \leftarrow$ create circular range search data structure on the points in $R(v)$.
20: **end for**

time to report k_i tuples (Lemma 5.3). Therefore, the overall query time is $O(\log n + k_F + k_F \log^2 n + k) = O(\log n + k \log^2 n)$, where $k = \sum_{b_i \in F} k_i$ is the output size and $k_F \leq k$. The space requirement for the first step in $O(n)$ while that of the second step is $O(n \log n)$. Therefore, the overall space requirement is $O(n \log n)$.

5.3.4 Query time improvement

The query time of our algorithm is dominated by the query time of the second step, specifically by the query time of T_R to explore red points for fruitful points and output tuples. Note that the query time of our algorithm can be improved by reducing the number of queries on T_R. Towards this end, we employ a technique similar to the one presented in Section 3.2.2 for the CTRQ problem. With each $b_i \in B$, we associate a list L_i of α-nearest red points ($\log n \leq \alpha \leq \log^2 n$) with score at least $\tau - s(b_i)$ sorted in

Algorithm 7 : Query

Input: DS_f: Data structure to find fruitful points; T_R: Data structure to form tuples; δ, τ: reals; q: query point.

Output: O: set of colored tuples satisfying the distance constraint δ with respect to q and score constraint τ.

1: $O \leftarrow \emptyset$
// `Step 1: Find fruitful points for` q
2: $D' \leftarrow$ Subset of disks D stabbed by q, as found by a disk stabbing query on DS_f with q.
3: $F \leftarrow$ Subset of blue points that are centers of disks in D'.
// F `is the set of fruitful blue points.`
// `Step 2: Query using fruitful points to form tuples`
4: **for all** $b_i \in F$ **do**
5: $Z_i \leftarrow [\tau - s(b_i), \infty)$
6: $\rho_i \leftarrow \delta - d(q, b_i)$
7: $R'_i \leftarrow \emptyset$
8: $\mathcal{V}_R(b_i) \leftarrow$ Set of canonical nodes of T_R that cover the range Z_i, as found by a range search.
9: **for all** $v \in \mathcal{V}_R(b_i)$ **do**
10: $R'_i \leftarrow R'_i \cup$ Subset of red points found by a circular range search on $DS_t(v)$ with b_i as query point (i.e., center) and radius ρ_i.
11: **end for**
12: **for all** $r_j \in R'_i$ **do**
13: $O \leftarrow O \cup \{(b_i, r_j)\}$
14: **end for**
15: **end for**
16: **return** O

non-decreasing order of distance from b_i. Let $\hat{r}_i$ represents the last red point in L_i.

To explore the red points for a fruitful point b_i, we first check $\hat{r}_i$ for the distance constraint. If $d(b_i, \hat{r}_i) > \delta - d(q, b_i)$, then b_i is part of less than α output tuples and we scan L_i from beginning and output (b_i, r_j) for each r_j that satisfies the condition $d(b_i, r_j) \leq \delta - d(q, b_i)$. Otherwise, b_i is part of at least α output tuples and we query T_R to find the tuples to output. By following an analysis similar to the one in Section 3.2.2, we get a solution with an overall query time of $O(\log n + \frac{k}{\alpha} \log^2 n)$ with $O(n\alpha)$ space. For example, if $\alpha = \log^2 n$ (resp., $\alpha = \log n$), we get a solution with $O(\log n + k)$ (resp., $O((1 + k) \log n)$) query time and $O(n \log^2 n)$ (resp., $O(n \log n)$) space.

We summarize the results in this section in the following theorem.

Theorem 5.1. *For a user-specified parameter $\alpha \in [\log n, \log^2 n]$, a fixed-distance fixed-threshold $(\tau, 2)$-CTRQ problem on a set of n red and blue points can be solved in $O(\log n + \frac{k}{\alpha}\log^2 n)$ time using $O(n\alpha)$ space, where k is the output size. In particular, $\alpha = \log n$ (resp., $\alpha = \log^2 n$) yields a solution with $O((k+1)\log n)$ (resp., $O(\log n + k)$) query time and $O(n \log n)$ (resp., $O(n \log^2 n)$) space.*

5.4 Fixed-distance variable-threshold $(\tau, 2)$-CTRQ

In this section, we discuss our solution for the fixed distance variable threshold version of the $(\tau, 2)$-CTRQ problem. As before, we present a two-step solution for this problem, where, in the first step, we find the set of fruitful points and, in the second step, we explore B and R using the fruitful points to form and report the tuples. However, the challenge now is that τ is not known at preprocessing time, so, the fruitful point-finding data structure of Section 5.3.1 cannot be used for this problem. Note that the second step does not rely on knowing τ at preprocessing time, so remains unchanged. In the rest of this section. we first present an alternate method to find the fruitful points for the fixed-distance variable-threshold $(\tau, 2)$-CTRQ problem, and then show the overall solution for the problem.

5.4.1 Identifying fruitful points

Recall that the fruitful points are the blue points that are guaranteed to be part of at least one output tuple with respect to a given q, δ, and τ. For each $b_i \in B$, let the *Pareto-optimal tuples of* b_i, denoted by $Pareto(b_i)$, be the subset of the tuples of $\{b_i\} \times R$ having the following properties.

1. For any $(b_i, r_k) \notin Pareto(b_i)$, there is a $(b_i, r_j) \in Pareto(b_i)$ such that $d(b_i, r_j) \leq d(b_i, r_k)$ and $s(b_i) + s(r_j) \geq s(b_i) + s(r_k)$; and
2. if $|Pareto(b_i)| > 1$ then for any two tuples $(b_i, r_j), (b_i, r_k) \in Pareto(b_i)$ either $d(b_i, r_j) < d(b_i, r_k)$ and $s(b_i) + s(r_j) < s(b_i) + s(r_k)$, or $d(b_i, r_j) > d(b_i, r_k)$ and $s(b_i) + s(r_j) > s(b_i) + s(r_k)$.

To compute $Pareto(b_i)$ for a b_i, we build a binary search tree T_i based on the scores of the points of R and their distances from b_i, as follows. Let $r_j \in R$ be the point with

highest score. (Ties are resolved by favoring the point with smallest distance to b_i.) At the root of T_i, we store (b_i, r_j). The left (resp., right) subtree is build recursively on the subset of points of R with distance to b_i at least (resp., smaller than) $d(b_i, r_j)$.

It is clear from the construction that the nodes of T_i have the following property. Let v_j and v_k be any two nodes of T_i, and let (b_i, r_j) and (b_i, r_k) be the tuples corresponding to v_j and v_k, respectively. If v_k is in the subtree rooted at v_j then $s(b_i) + s(r_j) \geq s(b_i) + s(r_k)$. Additionally, if v_k is in the subtree rooted at the left (resp., right) child of v_j, then $d(b_i, r_j) \leq d(b_i, r_k)$ (resp., $d(b_i, r_j) > d(b_i, r_k)$).

We start from the root and traverse the right branch of the tree. At any node, we add the corresponding (blue, red) tuple to an initially empty set $Pareto(b_i)$ and go to its right child. We stop the traversal if the search falls off the tree, i.e. the right child of the node does not exist. We call the above-traversed path the *right chain*.

We argue that $Pareto(b_i)$ computed as above has the desired properties. For any $(b_i, r_k) \notin Pareto(b_i)$, let v_k be the corresponding node in T_i. Clearly v_k is not a node in right chain. Let v_j be the last node of the right chain in the path from root to v_k and let $(b_i, r_j) \in Pareto(b_i)$ be the corresponding tuple. Therefore, according to the above-mentioned properties of T_i, $d(b_i, r_j) \leq d(b_i, r_k)$ and $s(b_i) + s(r_j) \geq s(b_i) + s(r_k)$, so Property 1 holds. Also, let (b_i, r) and (b_i, r') be tuples in $Pareto(b_i)$, stored at nodes v and v', of T_i, respectively. Clearly, v and v' belong to the right chain of T_i. If v' is in the subtree rooted at v then $d(b_i, r) > d(b_i, r')$ and $s(b_i) + s(r) > s(b_i) + s(r')$, and vice versa. Therefore, $Pareto(b_i)$ Property 2 also holds.

For a $b_i \in B$, the number of points in $Pareto(b_i)$ is the same as the length of the right chain of the binary search tree T_i, which is upper-bounded by the height of T_i. Even though, in the worst case, the height of T_i is $O(|R|)$, in the average case, i.e. when the scores are assigned uniformly at random, T_i is a *random binary search tree*, as noted in [72]. It has been shown that the height of such a tree is logarithmic in the number of points, i.e. $O(\log |R|) = O(\log n)$ [73]. Therefore, the expected size of $Pareto(b_i)$ is $O(\log n)$.

Lemma 5.4. *For any $b_i \in B$, let $Pareto(b_i)$ be the set of tuples as defined above. Let q be a query point, δ the query distance, and τ the query threshold. Then b_i is fruitful if and only if there is an $(b_i, r_j) \in Pareto(b_i)$ such that $d(q, b_i) + d(b_i, r_j) \leq \delta$ and $s(b_i) + s(r_j) \geq \tau$.*

Proof. ($\Leftarrow$) Suppose there is an $(b_i, r_j) \in \mathit{Pareto}(b_i)$ such that $d(q, b_i) + d(b_i, r_j) \leq \delta$ and $s(b_i) + s(r_j) \geq \tau$, therefore the tuple (b_i, r_j) is output. Hence, by definition, b_i is fruitful.

($\Rightarrow$) Suppose b_i is fruitful. Thus, there is an $r_k \in R$ such that (b_i, r_k) is output. Thus, $d(q, b_i) + d(b_i, r_k) \leq \delta$ and $s(b_i) + s(r_k) \geq \tau$. If $(b_i, r_k) \in \mathit{Pareto}(b_i)$ then we done. Otherwise, by definition there is an $(b_i, r_j) \in \mathit{Pareto}(b_i)$ such that $d(b_i, r_j) \leq d(b_i, r_k)$ and $s(b_i)+s(r_j) \geq s(b_i)+s(r_k)$, therefore $d(q, b_i)+d(b_i, r_j) \leq \delta$ and $s(b_i)+s(r_j) \geq \tau$. □

Based on Lemma 5.4 we propose the following preprocessing steps to help us identify the fruitful blue points at query time. First, for each $b_i \in B$, we compute $\mathit{Pareto}(b_i)$ as mentioned above. Next, for each $(b_i, r_j) \in \mathit{Pareto}(b_i)$, we create a weighted blue point, b_{ij} with weight $w(b_{ij}) = d(b_i, r_j)$ and score $s(b_{ij}) = s(b_i) + s(r_j)$. Let B' be the set of the newly created blue points. Now, by Lemma 5.4, for a query point q, a query distance δ, and a query threshold τ, b_i is fruitful if and only if there is a $b_{ij} \in B'$ such that $s(b_{ij}) \geq \tau$ and $d(q, b_i) + w(b_{ij}) \leq \delta$, i.e., $d(q, b_i) \leq \delta - w(b_{ij})$, i.e., q lies inside or "stabs" the disk D_{ij} centered at b_i with radius $\delta - w(b_{ij})$ and score $s(D_{ij}) \geq \tau$.

We build a 1-dimensional range tree $T_{B'}$ on the distinct values of the scores of the points in B', where the points of B' with same score are stored at a single leaf node. At any node $v \in T_{B'}$, let $B'(v)$ denote the points of B' stored at the subtree rooted at v (or at v, if v is a leaf node). At v, we build the disk stabbing data structure, $DS_f(v)$, of Section 5.3.1 on the points of $B'(v)$.

To find the fruitful points for a given q, (fixed) δ, and τ, we first query $T_{B'}$ with range $Z_\tau = [\tau, \infty)$ to find a set $\mathcal{V}_\tau$ of canonical nodes of $T_{B'}$ that cover the range Z_τ. Clearly, for each $v \in \mathcal{V}_\tau$, the score of each weighted blue point stored in the subtree rooted at v is at least τ. For each $v \in \mathcal{V}_\tau$, we query $DS_f(v)$ and report the blue points corresponding to the disks stabbed by q as fruitful points.

As mentioned before, the expected size of $\mathit{Pareto}(b_i)$ for any $b_i \in B$ is $O(\log n)$, and hence the expected size of $B' = O(n \log n)$. For a $v \in \mathcal{V}_\tau$, the disk stabbing data structure $DS_f(v)$ can be implemented in $O(|B'(v)|)$, therefore the expected size of $T_{B'}$ is $O(|B'| \log |B'|) = O(n \log^2 n)$.

For each $v \in \mathcal{V}_\tau$, the time to query $DS_f(v)$ is $O(\log |B'(v)| + k'_{Fv}) = O(\log n + k'_{Fv})$ time, where k'_{Fv} is the number of points reported when $DS_f(v)$ is queried. Since $|\mathcal{V}_\tau| = O(\log n)$, the total time required to find the fruitful points is $O(\log^2 n + k'_F)$, where

$k'_F = \sum_{v \in \mathcal{V}_\tau} k'_{Fv}$ is the total number of fruitful points reported. Note that even though a blue point b_i can be reported multiple times, each instance of b_i corresponds to an output tuple. Therefore, the cost of reporting duplicate blue points can be charged to the final output tuples of the fixed-distance variable-threshold $(\tau, 2)$-CTRQ problem.

Lemma 5.5. *Let B be a set of blue point and R be a set of red points, where $|B|+|R| = n$. There exists a data structure of expected size $O(n \log^2 n)$ which can report the k'_F points of B that are fruitful with respect to q, δ, and τ in $O(\log^2 n + k'_F)$ time.*

5.4.2 Overall algorithm

We now present the overall algorithm for the fixed-distance variable-threshold $(\tau, 2)$-CTRQ problem. Recall that the first step of the algorithm is to find the fruitful blue points with respect to q, δ, and τ, as discussed above. The second step is to explore red points from each fruitful point to form and report the desired (blue, red) tuples.

The second step is identical to that of the fixed threshold version of the problem. Note that the approach mentioned in Section 5.3.4 for improving the query time is not applicable here as it requires τ to be known during preprocessing. Lemma 5.3 summarizes the bounds for the second step for a given fruitful point.

As noted above, the set of fruitful points computed in the first step may have duplicates. To avoid unnecessary repeated execution of the second step for the same fruitful point, we maintain a flag with each blue point. For a fruitful point, we check the flag. If the flag is not set, we execute the second step and set the flag. Otherwise, we skip the fruitful point. We reset the flags at the end of the query algorithm. Hence, the overall runtime for the second step, takes over all fruitful points, is $O(k_F \log^2 n + k'_F + k) = O(k \log^2 n)$, where k_F, k'_F, and k are the number of fruitful point, number of points reported in the first step, and total output size, respectively. (Note that $k_F \leq k'_F \leq k$.) By combining the bounds for first and second step, we get following result for the fixed-distance variable-threshold $(\tau, 2)$-CTRQ problem.

Theorem 5.2. *A fixed-distance variable-threshold $(\tau, 2)$-CTRQ problem on a set of n red and blue points can be solved in $O((k+1) \log^2 n)$ time using a data structure of expected size $O(n \log^2 n)$, where k is the output size.*

5.5 Handling variable distance queries

In this section, we discuss the changes required to extend our solutions for the two fixed distance versions of the $(\tau, 2)$-CTRQ problem (Section 5.3 and 5.4) to the corresponding variable distance versions. Again, $\tau \geq 0$ can be fixed of variable. Note that the second step, i.e. exploring red points for each fruitful blue point and forming output tuples does not depend on δ being known at preprocessing, and hence remains same. We need only need to update the first step to handle the variable distances.

It is easy to see that the approaches underlying Lemma 5.1 and Lemma 5.4 do not depend on δ being fixed, and hence are applicable to the variable-distance versions of the fixed-threshold and variable-threshold problems, respectively. Therefore, as before, we compute the set of weighted blue points from the given sets of red and blue points. Specifically, for the fixed threshold version of the problem, we apply Lemma 5.1 to compute nearest red point $r_j \in R$ for each $b_i \in B$ such that $s(r_j) \geq \tau - s(b_i)$ and associate its distance from b_i as a weight $w(b_i)$. For the variable threshold version of the problem, we apply Lemma 5.4 to first compute set $Pareto(b_i)$ for each $b_i \in B$ and then compute the weighted point-set B' as shown in Section 5.4.1.

Instead of building the fruitful point-finding structure of the fixed distance 2-CTRQ problem (i.e., the disk stabbing structure) on this weighted point-set, we build the fruitful point-finding structure for the variable distance 2-CTRQ problem using the transformation-based or the weighted Voronoi diagram-based method (Section 3.3).

Theorem 5.3 and Theorem 5.4 below combine the bounds for the corresponding first and second steps and list the overall bounds for the variable-distance version of the fixed threshold and variable threshold problem, respectively. (Only the bounds which are not worse than others in both runtime and space have been shown.)

Theorem 5.3. *A variable-distance fixed-threshold* $(\tau, 2)$-CTRQ *problem on a set of* n *red and blue points can be solved in* (i) $O(\sqrt{n}\log^{O(1)} n + \frac{k}{\alpha}\log^2 n)$ *time using* $O(n\alpha)$ *space, where* $\alpha \in [\log n, \log^2 n]$, *or* (ii) $O((1+k)\log n)$ *time using* $O(n\log^2 n)$ *space, or* (iii) $O(\log n + k)$ *time using* $O(n\log^2 n)$ *space. Here* k *is the output size. (The bounds in* (iii) *involve a data structure that is built probabilistically in preprocessing; the query time is deterministic.)*

Theorem 5.4. *A variable-distance variable-threshold* $(\tau, 2)$-CTRQ *problem on a set of* n *red and blue points can be solved in* (i) $O(\sqrt{n}\log^{O(1)} n + k\log^2 n)$ *time using* $O(n\log^2 n)$ *expected space, or* (ii) $O((1+k)\log^2 n)$ *time using* $O(n\log^3 n)$ *expected space. Here* k *is the output size.*

5.6 Handling more than two colors

In this section, we generalize our approach for the $(\tau, 2)$-CTRQ problem to solve the general τ-CTRQ problem, i.e. the problem with $m > 2$ colored points (Problem 5.1). As in the case of $(\tau, 2)$-CTRQ, here, too, $\delta > 0$ and $\tau \geq 0$ can both be (independently) fixed or variable, resulting in four versions of the problem.

Similar to the $(\tau, 2)$-CTRQ problem, we present a two-step solution for the τ-CTRQ problem, where in the first step we find the fruitful points of C_1, and in the second step, we explore from each fruitful point the points of $C_2, \ldots, C_m$, in that order, to form and report the desired tuples. Here a point of C_1 is *fruitful* if it is part of at least one output tuple for query point q, query distance δ, and query threshold τ. However, the challenge here is that for $m > 2$ colors, Lemma 5.1 and Lemma 5.4 cannot be applied directly to assign weights to the points of C_1. Also, circular range search cannot be used in the second step to explore the points of $C_2, \ldots, C_m$ to form and report the tuples. In the rest of this section we discuss how to overcome these challenges.

Let $CS_i = (c_i, \ldots, c_m)$, for $i \in [1, m]$, be the truncated ordering of CS starting from the ith color. We define the *Pareto-optimal tuples of* p_i, i.e., $Pareto(p_i)$, for each $p_i \in C_i$ as the subset of the tuples of $\{p_i\} \times C_{i+1} \times \ldots \times C_m$ having the following properties.

1. For any tuple $t' \notin Pareto(p_i)$, there is a tuple $t \in Pareto(p_i)$ such that $l_{path}(t) \leq l_{path}(t')$ and $s(t) \geq s(t')$; and

2. if $|Pareto(p_i)| > 1$ then for any two tuples $t, t' \in Pareto(p_i)$ either $l_{path}(t) < l_{path}(t')$ and $s(t) < s(t')$, or $l_{path}(t) > l_{path}(t')$ and $s(t) > s(t')$.

The following lemma provides a useful characterization of the tuples in $Pareto(p_1)$ for any $p_1 \in C_1$.

Lemma 5.6. *For any $p_1 \in C_1$, let $Pareto(p_1)$ be the set of Pareto-optimal tuples as defined above. Let $t = (p_1, \ldots, p_m)$ be a tuple in $Pareto(p_1)$. Then for the point $p_i \in C_i$, $i \in [2, m]$, $t_i = (p_i, \ldots, p_m)$ is in $Pareto(p_i)$.*

Proof. Let us assume that $t_i \notin Pareto(p_i)$. Then by definition there is a tuple $t'_i = (p_i, p'_{i+1}, \ldots, p'_m) \in Pareto(p_i)$ such that $l_{path}(t'_i) < l_{path}(t_i)$ and $s(t'_i) > s(t_i)$. Let $\hat{t} = (p_1, \ldots, p_i, p'_{i+1}, \ldots, p'_m)$. Then

$$l_{path}(\hat{t}) = \sum_{k=2}^{i} d(p_{k-1}, p_k) + l_{path}(t'_i) \quad < \quad \sum_{k=2}^{i} d(p_{k-1}, p_k) + l_{path}(t_i) = l_{path}(t).$$

Also,

$$s(\hat{t}) = \sum_{k=1}^{i} s(p_k) + \sum_{k=i+1}^{m} s(p'_k) \quad > \quad \sum_{k=1}^{i} s(p_k) + \sum_{k=i+1}^{m} s(p_k) = s(t).$$

This contradicts the assumption that $t \in Pareto(p_1)$. □

Based on Lemma 5.6 we compute the desired set of tuples $Pareto(p_1)$ for each $p_1 \in C_1$ as follows. We process the sets $C_{m-1}, C_{m-2}, \ldots, C_1$, in that order, and compute the Pareto-optimal tuples for the points of each set. For any point $p_i \in C_i$, $i \in [1, m-1]$, we use the approach of Section 5.4.1 to create a binary search tree on the Pareto-optimal tuples of all points in C_{i+1}. Next, we add each tuple t_k on the right path to the set $Pareto(p_i)$.

Lemma 5.7. *For any $p_1 \in C_1$, let $Pareto(p_1)$ be the set of Pareto-optimal tuples as defined above. Let q be a query point, δ the (fixed or variable) query distance, and τ the (fixed or variable) query threshold. Then p_1 is fruitful if and only if there is a tuple $t \in Pareto(p_1)$ such that $d(q, p_1) + l_{path}(t) \leq \delta$ and $s(t) \geq \tau$.*

Proof. ($\Leftarrow$) Suppose there is a $t \in Pareto(p_1)$ such that $d(q, p_1) + l_{path}(t) \leq \delta$ and $s(t) \geq \tau$. Then the tuple t is output. Hence, by definition, p_1 is fruitful.

($\Rightarrow$) Suppose p_1 is fruitful. Thus, there is a t' of CS such that t' is output. Thus, $d(q, p_1) + l_{path}(t') \leq \delta$ and $s(t') \geq \tau$. If $t' \in Pareto(p_1)$ then we done. Otherwise, by definition there is a $t \in Pareto(p_1)$ such that $l_{path}(t) \leq l_{path}(t')$ and $s(t) \geq s(t')$, therefore $d(q, p_1) + l_{path}(t) \leq \delta$ and $s(t) \geq \tau$. □

Lemma 5.7 above suggests the following preprocessing steps to help determine the fruitful points of C_1 during query time. For each $p_1 \in C_1$, we compute $Pareto(p_1)$ as shown above. For each $t_j \in Pareto(p_1)$, we create a weighted point p_{1j} with weight $w(p_{1j}) = l_{path}(t_j)$ and score $s(p_{1j}) = s(t_j)$. Let C_1' be the set of newly created points. With this weight assignment the problem of finding fruitful points of C_1 for a q, δ, and τ becomes finding weighted points $p_{1j} \in C_1'$ such that $d(q, p_{1j}) + w(p_{1j}) \leq \delta$ and $s(p_{1j}) \geq \tau$. This is similar to the case for $m = 2$ colors. Therefore, we can use the solution presented in Section 5.3, Section 5.4, or Section 5.5 for the corresponding version of the τ-CTRQ problem. Note that for the fixed threshold versions of the problem, it is sufficient to create only one weighted point for each $p_1 \in C_1$ corresponding to the tuple $t_j \in Pareto(p_1)$ with smallest $l_{path}(t_j)$ such that $s(t_j) \geq \tau$. The proof of this claim is trivial and therefore omitted.

Recall that the second step is to explore the points of color $c_2, \ldots, c_m$ to form and report the tuples, i.e., for each fruitful point $p_1 \in C_1$, the goal is to identify an $(m-1)$-tuple $t = (p_2, \ldots, p_m)$ such that $s(p_1) + s(t) \geq \tau$ and $d(q, p_1) + d(p_1, p_2) + l_{path}(t) \leq \delta$, i.e., $s(t) \geq \tau - s(p_1)$ and $d(p_1, p_2) + l_{path}(t) \leq \delta - d(q, p_1)$, i.e., $p_2 \in C_2$ is fruitful point for the "query" point p_1, query distance $\delta - d(q, p_1)$, and query threshold $\tau - s(p_1)$ with color sequence $CS_2 = (c_2, \ldots, c_m)$. (Note that p_1 is an input point that acts as a "query" point here.) Hence, for the second step, we recursively solve the τ-CTRQ problem using the fruitful points as "query" points with query distance, query threshold, and color sequence adjusted appropriately. Note that, even if δ and τ are known for the original problem, the query distance (e.g., $\delta - d(q, p_1)$) and the query threshold (e.g., $\tau - s(p_1)$) are not known beforehand for the successive queries, therefore the recursive problem is a variable-distance variable-threshold τ-CTRQ problem. We terminate the recursion when only two colors are left to process. At this point, we use the solution for the variable-distance variable-threshold $(\tau, 2)$-CTRQ problem. We form the tuples as we backtrack and report the tuples so formed as the answer set.

Analysis. Based on the query distance and query threshold being fixed or variable, we have four versions of the problem. We analyze only the two variable distance versions of the problem, namely the variable-distance fixed-threshold and the variable-distance variable-distance τ-CTRQ problem. The corresponding fixed distance versions of the

problem can be analyzed in a similar fashion.

Let $Q_{fix}(m,n)$ (resp., $Q_{var}(m,n)$) be the query time of the variable-distance fixed-threshold (resp., variable-threshold) τ-CTRQ problem on m colors and n points. Also, let $Q_{F,fix}(n)$ (resp., $Q_{F,var}(n)$) be the time required to find the fruitful points in a set of n points for the fixed (resp., variable) threshold version of the problem. For $m \geq 3$ colors, the above approach leads to the following recurrence relation.

$$Q_{fix}(m,n) = Q_{F,fix}(n) + k_{F_1} \times Q_{var}(m-1,n), \tag{5.1}$$

$$Q_{var}(m,n) = Q_{F,var}(n) + k_{F_1} \times Q_{var}(m-1,n), \tag{5.2}$$

where k_{F_1} is the number of fruitful points of C_1 with respect to q, δ, and τ. The base case for the above recurrence is $m = 3$. For $m = 3$, the query time for the variable-distance fixed-threshold τ-CTRQ problem is

$$Q_{fix}(3,n) = Q_{F,fix}(n) + k_{F_1} \times Q_{var}(2,n).$$

We use the results mentioned in Lemma 5.1 and Theorem 5.4 for $Q_{var}(2,n)$ and $Q_{F,fix}(n)$, respectively, to derive the corresponding time and space bounds for $Q_{fix}(3,n)$. For example, using $Q_{var}(2,n) = O((1+k)\log^2 n)$ (with expected space usage $O(n\log^3 n)$) and $Q_{F,fix}(n) = O(\log n + k_F)$ (with space usage $O(n\log^2 n)$), we get

$$Q_{fix}(3,n) = O(\log n + k_{F_1}) + k_{F_1} O(\log^2 n) + O(\log^2 n) \sum_{i=1}^{k_{F_1}} k_i = O(\log n + k\log^2 n).$$

Here, k_i is the number of output tuples with the ith fruitful point in them, $\sum_{i=1}^{k_{F_1}} k_i = k$. (Note that the product of the number of fruitful points of each color is at most the output size k, and hence, the number of fruitful points of each color is also at most k.) The space usage is bounded by the space usage of the variable-distance variable-threshold $(\tau,2)$-CTRQ problem. Therefore the space usage of the fixed threshold version of τ-CTRQ problem for $m = 3$ is $O(n\log^3 n)$. Note that this space bound is an expected bound. Other query times and space bounds for the variable-distance versions of the τ-CTRQ problem for $m = 3$ can be derived similarly.

Using $m = 3$ as the base case for the recurrence relation in Equations 5.1 and 5.2, the space and time bounds for the variable-distance fixed-threshold (resp., variable-threshold) τ-CTRQ problem can be verified to be as listed in Theorem 5.5 (resp.,

Theorem 5.6). (Only the bounds which are not worse than others in both runtime and space have been shown.)

Theorem 5.5. *A variable-distance fixed-threshold* τ-CTRQ *problem on a set of* n *points in the plane, where each point is assigned a color* c_i *from a pallet of* m *colors* ($m \geq 2$)*, can be solved in either* (*i*) $O(\log n + km\sqrt{n}\log^{O(1)} n)$ *time using* $O(n \log^2 n)$ *expected space data structure, or* (*ii*) $O(\log n + km \log^2 n)$ *time using an* $O(n \log^3 n)$ *expected space data structure.*

Theorem 5.6. *A variable-distance variable-threshold* τ-CTRQ *problem on a set of* n *points in the plane, where each point is assigned a color* c_i *from a pallet of* m *colors* ($m \geq 2$)*, can be solved in either* (*i*) $O((1 + km)\sqrt{n}\log^{O(1)} n)$ *time using* $O(n \log^2 n)$ *expected space data structure, or* (*ii*) $O((1+km)\log^2 n)$ *time using an* $O(n \log^3 n)$ *expected space data structure.*

By an analysis similar to the one for the two fixed distance versions of the problem, it is easy to verify that the space and time bounds for the fixed-distance fixed-threshold (resp., variable-threshold) τ-CTRQ problem are as listed in Theorem 5.7 (resp., Theorem 5.8). (Again, only the bounds which are not worse than others in both runtime and space have been shown.)

Theorem 5.7. *A fixed-distance fixed-threshold* τ-CTRQ *problem on a set of* n *points in the plane, where each point is assigned a color* c_i *from a pallet of* m *colors* ($m \geq 2$)*, can be solved in either* (*i*) $O(\log n + km\sqrt{n}\log^{O(1)} n)$ *time using* $O(n \log^2 n)$ *expected space data structure, or* (*ii*) $O(\log n + km \log^2 n)$ *time using an* $O(n \log^3 n)$ *expected space data structure.*

Theorem 5.8. *A variable-distance variable-threshold* τ-CTRQ *problem on a set of* n *points in the plane, where each point is assigned a color* c_i *from a pallet of* m *colors* ($m \geq 2$)*, can be solved in either* (*i*) $O(\log^2 n+km\sqrt{n}\log^{O(1)} n)$ *time using* $O(n \log^2 n)$ *expected space data structure, or* (*ii*) $O((1+km)\log^2 n)$ *time using an* $O(n \log^3 n)$ *expected space data structure.*

Chapter 6

Colored Tuple Range Skyline Query (SkyCTRQ)

For the colored point-sets where each point has a single real-values feature attribute value, in this chapter, we extend the CTRQ problem to report the tuples that satisfy the distance constraint and are not dominated by any other tuples that satisfy the distance constraint (Section 1.2.3.3). Here, a tuple t_i dominates another tuple t_j if and only if the feature attribute value of each point in t_i is larger than or equal to the feature attribute value of the point of the same color in t_j, with strict inequality for at least one point. We call this the *colored tuple range skyline query* (SkyCTRQ) problem. This is a generalization of the range-skyline query problem to point-tuples. We first present several efficient solutions for a Halfspace-Range-Skyline query problem, where the goal is to report the skyline (computed in feature space) of points that lie in a query halfspace. Next, we show how to efficient transform the SkyCTRQ problem it to the Halfspace-Range-Skyline query problem.

6.1 Problem formulation

Let $B = \{b_1, \ldots, b_{n_1}\}$ be a set of blue points and let $R = \{r_1, \ldots, r_{n_2}\}$ be a set of red points in a plane ($n_1 + n_2 = n$), where each point $p \in B \cup R$ has a single associated real-valued feature attribute, denoted by $a(p)$. For any $b_i, b_k \in B$ and $r_j, r_l \in R$, we say tuple (b_i, r_j) *dominates* tuple (b_k, r_l), denoted by $(b_i, r_j) \succ (b_k, r_l)$, if and only if

$a(b_i) \geq a(b_k)$ and $a(r_j) \geq a(r_l)$ with strict inequality in at least one condition.

Problem 6.1 (SkyCTRQ)**.** *Let the sets B and R be as defined above. Preprocess the sets B and R so that for any query point q in the plane and a distance δ, we can report all tuples (b_i, r_j) that satisfy the distance constraint $d(q, b_i) + d(b_i, r_j) \leq \delta$ and are not dominated by any other tuple satisfying the same distance constraint.*

Depending on the application there are two versions on the problem that are of interest. In the first version $\delta > 0$ is known during preprocessing while in the second version δ is known only at the query time. We refer to these versions as the *fixed distance* and the *variable distance* problems, respectively.

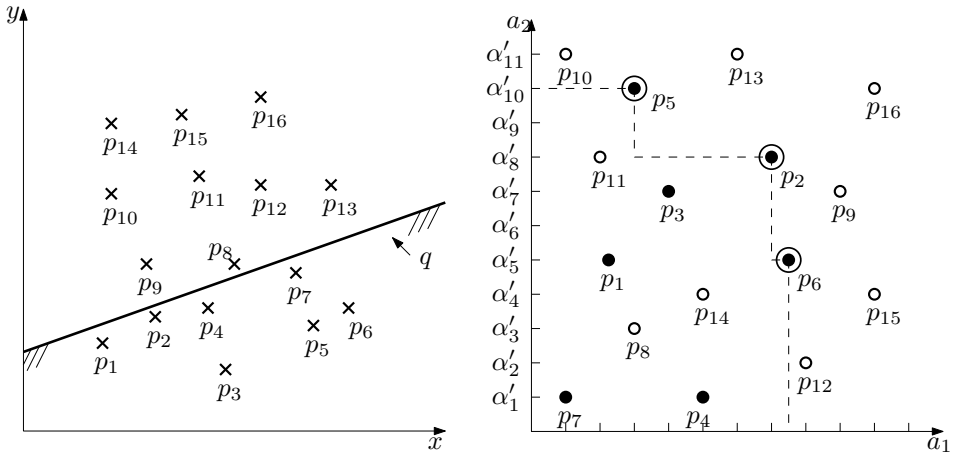
We will solve the SkyCTRQ problem by transforming it to a Halfspace-Range-Skyline query problem. Throughout, we will first present efficient algorithms for various instances of the Halfspace-Range-Skyline query problem, and then in Section 6.6 we will show how to transform the SkyCTRQ problem to an instance of the Halfspace-Range-Skyline query problem. We define the Halfspace-Range-Skyline query problem as follows.

Let S be a set of n points in a d-dimensional *coordinate space*, where each point $p \in S$ has $t \geq 2$ additional real-valued attributes, called *features*. We call $\mathbb{R}^t$ the *feature space*. Let the ith coordinate value and the jth feature value of a point $p \in S$ be denoted by $x_i(p)$ and $a_j(p)$, respectively. We say that point $p_i \in S$ *dominates* point $p_j \in S$ (denoted by $p_i \succ p_j$) if and only if $a_k(p_i) \geq a_k(p_j)$ for $1 \leq k \leq t$, with strict inequality for at least one feature value. The *skyline* of S is the subset of points of S in which no point dominates another.

Problem 6.2 (Halfspace-Range-Skyline)**.** *Preprocess S into a suitable data structure so that for any query halfspace q in $\mathbb{R}^d$ (coordinate space), we can report the skyline of $q \cap S$ (computed in feature space $\mathbb{R}^t$).*

For example, consider the set, S, of points with $t = 2$ in 2-dimensional coordinate space as shown in Figure 6.1. In Figure 6.1a, the hatched area denotes the query halfspace q. As shown in Figure 6.1b, the answer to the Halfspace-Range-Skyline query consists of points p_2, p_5, and p_6.

Note that the Halfspace-Range-Skyline problem for points with $t = 1$ or $d = 1$ can be solved efficiently using existing solutions, and hence is not considered here.



(a) Coordinate space. (b) Feature space.

Figure 6.1: An example of the HALFSPACE-RANGE-SKYLINE query. (a) A set S of points in 2-dimensional coordinate space, each with two features (not shown). Here, q (indicated by the hatched area) is the query halfspace. (b) Points of S in the 2-dimensional feature space. Here, the points of $q \cap S$ are shown solid ($\bullet$), points not in $q \cap S$ are shown unfilled ($\circ$), and the points in the answer set are circled.

Specifically, in 1-dimensional feature space, the skyline contains only the point with highest feature attribute value. Therefore, for points with $t = 1$, the HALFSPACE-RANGE-SKYLINE query problem is equivalent to an instance of top-k halfspace range search problem [74] with $k = 1$. Also, in 1-dimensional coordinate space, a negative (resp., positive) halfspace is just a *halfline* $[q, -\infty)$ (resp., $[q, +\infty)$). Therefore, for points with $d = 1$, the HALFSPACE-RANGE-SKYLINE query problem can be solve using the solution for 1-dimensional orthogonal range-skyline problem [21] with suitable query range.

We use $HSSky(S, q)$ to denote the answer set of the HALFSPACE-RANGE-SKYLINE query on S with a query halfspace q. Where there is no confusion, we will use q interchangeably to denote both the query halfspace and the defining hyperplane of the query halfspace.

6.2 Contributions

In this chapter, we present the following results for the Halfspace-Range-Skyline query problem and the SkyCTRQ problem.

- In Section 6.4, we discuss our approach for the Halfspace-Range-Skyline query problem for points with $t = 2$ feature attributes in d-dimensional coordinate space ($d \geq 2$). We present two data structures with different space and query time bounds to implement our approach. Specifically, for $d = 2, 3$, our first data structure uses $O(n \log n)$ space and has a query time of $O((1 + k) \log^2 n)$, while the second data structure uses $O(n^{1+\varepsilon} \log \log n)$ space and has a query time of $O((1 + k) \log n)$, where k is the output size. (The space and query-time bounds for $d \geq 4$ are listed in Theorem 6.1.) Both the data structures are based on a partition tree [2].

- In Section 6.5, we solve the Halfspace-Range-Skyline query problem for points with $t \geq 3$ feature attributes in d-dimensional coordinate space ($d = 2, 3$). For $d = 2$ (resp., $d = 3$), the algorithm uses $O(n^2 \log n \log \log n)$ (resp., $O(n^2 \log^2 n \log \log n)$) space and the query time is $O(\log^3 n + k)$ (resp., $O(n^{1/2+\varepsilon} \log n + k)$). The algorithm is based on a geometric transformation that maps the problem to a certain generalized non-intersection query problem [75].

- In Section 6.6, we present the SkyCTRQ problem as an application of the Halfspace-Range-Skyline query problem. We show that the SkyCTRQ problem can be mapped to an instance of the Halfspace-Range-Skyline problem via a certain geometric transformation. We present efficient solutions for the fixed and variable distance versions of the SkyCTRQ problem. For instance, for the fixed distance version, the algorithm takes $O(n \log^2 n)$ (resp., $O((n \log n)^{1+\varepsilon} \log \log n)$) expected space and has a query time of $O((1 + k) \log^2 n)$ (resp., $O((1 + k) \log n)$). (The bounds for the variable distance version of the problem are listed in Theorem 6.4.)

6.3 Preliminaries

Partition tree. The partition tree, originally proposed by Willard [76], is a well-studied and commonly used hierarchical data structure for halfspace range searching and simplex range searching problems [2, 35]. (See [77] for a comprehensive survey on the partition tree.) For a set S of n points in $\mathbb{R}^d$, a *simplicial partition* of S is defined as a set $\Pi = \{(S_1, \Delta_1), (S_2, \Delta_2), \ldots, (S_r, \Delta_r)\}$, where S_i's are disjoint subsets of S, and Δ_i is a d-dimensional simplex containing S_i. Note that the Δ_i's are not required to be disjoint, so a point of S may lie in many simplexes but it belongs to only one subset. A simplicial partition is called *fine* or *balanced* if each subset contains at most twice the average number of points of the subsets, i.e., $|S_i| \leq 2n/r$ for each $1 \leq i \leq r$.

Using simplicial partitions, a partition tree T on S is built as follows. Let $r = n^{\varepsilon}$ for a sufficiently small constant $\varepsilon > 0$. Each node v of T is associated with a subset $S_v \subseteq S$ and a simplex Δ_v. (If v is not a leaf node, then the points of S_v are not explicitly stored at v.) The root of T is associated with $S_{root} = S$ and $\Delta_{root} = \mathbb{R}^d$. If $|S_v|$ is a small constant then v is a leaf node and we store the points of S_v at v. Otherwise, v is an internal node with r children, each corresponding to a subset of a fine simplicial partition for S_v. At each child of v, we recursively construct a partition tree on the corresponding subset of points (Figure 6.2).

For a query halfspace q, we can answer a halfspace range search query, i.e., report the points in $q \cap S$, as follows. We explore the partition tree recursively starting at the root. When visiting a node v, if v is leaf node then we report the points of S_v that lie in q. Otherwise, for each child u of v, we test Δ_u against q. If Δ_u lies completely inside q, we report all points stored in the subtree rooted at u; if Δ_u lies completely outside q, we ignore u; finally, if q intersects Δ_u, we recursively explore the node u. It has been shown in [35] that a partition tree on n points in $\mathbb{R}^d$ requires $O(n)$ space and has a halfspace range query time of $O(\log n + k)$ for $d = 2, 3$ and $O(n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n + k)$ for $d \geq 4$, where k is the output size.

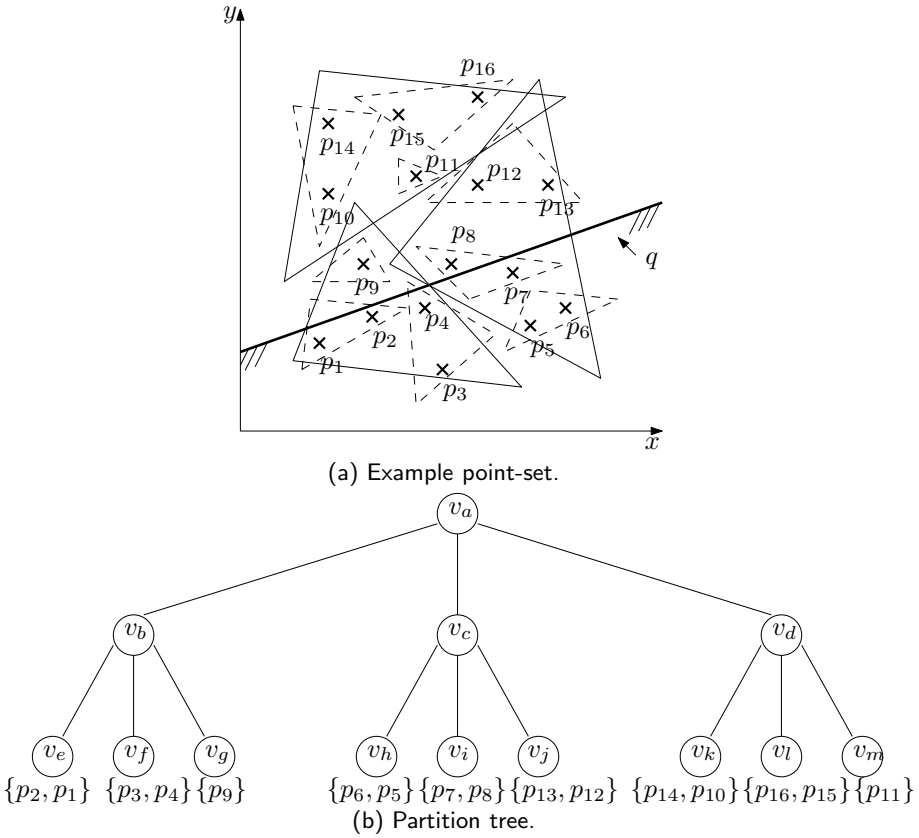


Figure 6.2: An example construction of the partition tree data structure for a point set in $\mathbb{R}^2$. (a) A hierarchical simplicial partition of points. Here, q is the query halfspace. (b) The partition tree.

6.4 Halfspace-Range-Skyline for points with $d \geq 2$ and $t = 2$

In this section, we present our solution for the Halfspace-Range-Skyline query problem for points with $t = 2$ feature attributes in d-dimensional coordinate space ($d \geq 2$). Note that even though the halfspace range query is executed in the d-dimensional coordinate space, the skyline is computed in the 2-dimensional feature space. In the latter space, there exists a total order among the points of $HSSky(S, q)$ based on their

feature attribute values. (If the points are sorted by $a_1(\cdot)$ values then they are reverse-sorted by $a_2(\cdot)$ values, and vice versa.) Our solution takes advantage of this total order and reports the points in the decreasing order of their a_1-feature attribute value. At a high level, our solution first reports a point that is guaranteed to be the point in $\mathit{HSSky}(S, q)$ with highest a_1-feature attribute value, and then prunes away the subset of points that is dominated by it. The algorithm solves the problem recursively on the remaining set of points. The recursion stops when no point is left to be explored.

Now we formally describe our solution. For simplicity in discussion, throughout we focus on the points of S that lie in the query halfspace q, i.e., the points of $q \cap S$. Let $p'_1, \ldots, p'_K$ be the list of these points, sorted in the decreasing order of their a_1-feature attribute values (ties broken in favor of the point with highest a_2-feature attribute value). Clearly, p'_1 is not dominated by any point in $q \cap S$, so we report it. Now observe that for any point $p'_j \in q \cap S$, if $a_2(p'_j) \leq a_2(p'_1)$ then $p'_1 \succ p'_j$, and hence $p'_j \notin \mathit{HSSky}(S, q)$. Therefore, the points of $q \cap S$ with a_2-feature attribute values at most $a_2(p'_1)$ can be pruned away from consideration. Thus, to search for the next point in $\mathit{HSSky}(S, q)$, we restrict our attention to the points with a_2-feature value larger than $a_2(p'_1)$. Let p'_i be the first point in the sorted list after p'_1 such that $a_2(p'_i) > a_2(p'_1)$.

We claim that p'_i is the next point in $\mathit{HSSky}(S, q)$. The intuition behind this claim is as follows. By definition, $a_2(p'_i) > a_2(p'_j)$ for any $j \in [1, i-1]$. Also, $a_1(p'_i) \geq a_1(p'_j)$ for any $j \in [i, K]$. Therefore, $p'_j \not\succ p'_i$ for any $j \in [1, K]$, and hence $p'_i \in \mathit{HSSky}(S, q)$. Based on this claim, we solve the problem recursively on the subset of points of $q \cap S$ with a_2-feature value larger than $a_2(p'_1)$. We stop the recursion when no more points are left to be explored.

Algorithm 8 shows our general approach for the Halfspace-Range-Skyline query problem for points with $t = 2$ feature attributes in d-dimensional coordinate space ($d \geq 2$). In what follows, we describe two data structures with different space usage and query time bounds to implement our algorithm. Specifically, we first show a low space data structure based on a certain combination of the range tree and the partition tree data structures [2]; however, the query time is somewhat high. We then improve the query time (with some space penalty) by suitably augmenting the partition tree.

Algorithm 8 : HALFSPACE-RANGE-SKYLINE(S, q)

Input: S: set of input points with $t = 2$ feature attributes in $\mathbb{R}^d$ coordinate space; q: query halfspace in $\mathbb{R}^d$ coordinate space.

Output: $HSSky(S, q)$: Skyline of the points of $q \cap S$ in $\mathbb{R}^2$ feature space.

1: $HSSky(S, q) \leftarrow \emptyset$
2: $\hat{S} \leftarrow S$
3: **while** $\hat{S} = \emptyset$ **do**
4: $\quad p \leftarrow$ The point of $q \cap S$ with highest a_1-feature value (ties resolved in favor of the point with highest a_2-feature value).
5: $\quad HSSky(S, q) \leftarrow HSSky(S, q) \cup \{p\}$
6: $\quad \hat{S} \leftarrow$ The points of $\hat{S}$ with a_2-feature values in the range $(a_2(p), \infty)$.
7: **end while**

First data structure. The points of S are organized in a 2-level tree structure as follows. We first build a range tree, T, based on the distinct a_2-feature attribute values of the points in S. At a leaf node v corresponding to some value α, we store the points of S with the a_2-feature attribute value equal to α. At any node $v \in T$, let $S(v)$ be the set of points stored at the subtree rooted at v (or at v, if v is a leaf node). At v, we build a partition tree, $T_{hs}(v)$, in the coordinate space on the points in $S(v)$ (Section 6.3). We augment each node $u \in T_{hs}(v)$ with a point $p_{max}(S(u))$ that has the largest a_1-feature value (ties broken in favor of the point with the highest a_2-feature value) stored in the subtree rooted at u (Figure 6.3).

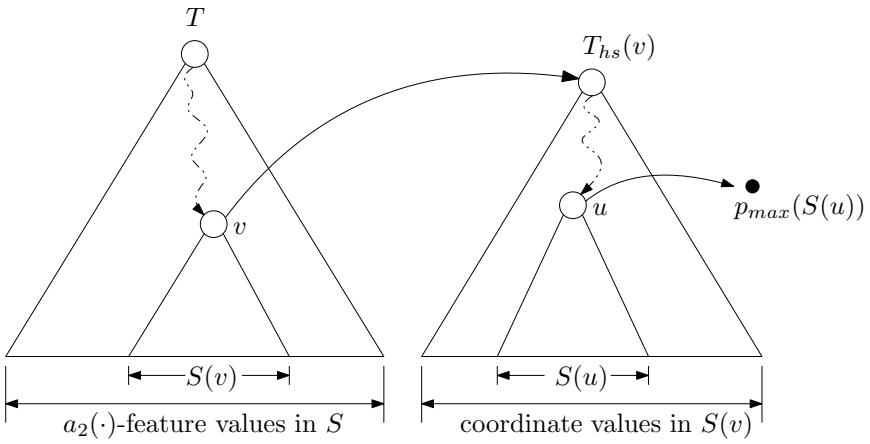


Figure 6.3: An illustration of our first data structure.

Let $Z_0 = (-\infty, \infty)$ and $HSSky(S, q) = \emptyset$. To answer a HALFSPACE-RANGE-SKYLINE query for a given query halfspace q, we iterate on the following steps. On the ith iteration, $i \geq 1$, we first query T with the range Z_{i-1} and get a set of canonical nodes, $\mathcal{V}_i$, that cover the range Z_{i-1} [2]. Clearly, if $|\mathcal{V}_i| = 0$ then there is no point with a_2-feature value in the range Z_{i-1}, hence we terminate the algorithm. Otherwise, $|\mathcal{V}_i| > 0$ and for each $v \in \mathcal{V}_i$, we query the associated partition tree, $T_{hs}(v)$, with q and find a set of canonical nodes, $\mathcal{V}_{hs}(v)$, that cover the query halfspace range [35]. Let $\mathcal{V} = \bigcup_{v \in \mathcal{V}_i} \mathcal{V}_{hs}(v)$. We check the associated $p_{max}(S(u))$ for each $u \in \mathcal{V}$ to find the point, p, with largest a_1-feature value (again, ties broken in favor of the point with highest a_2-feature value). (If v is a leaf node then we check all the points in stored in v to get $p_{max}(\cdot)$. Since each leaf node contains constant number of points, check each point does not impact the asymptotic runtime.) Clearly, p is a skyline point. We add p to the set $HSSky(S, q)$ and set $Z_i = (a_2(p), \infty)$. We terminate the algorithm, if no point is reported in the current iteration.

The query time and the space required for our data structure T depends on the query time and the space required for the partition tree. As mentioned in Section 6.3, a partition tree on n points in d-dimensional coordinate space require $O(n)$ space. Also, it is easy to see from the discussion in Section 6.3 that the canonical nodes that cover a query halfspace range can be found in $O(\log n)$-time for $d = 2, 3$ and $O(n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n)$-time for $d \geq 4$. Augmenting each node of the partition tree with a point with largest a_1-feature value takes constant space per node. Therefore, for any $v \in T$, $T_{hs}(v)$ takes $O(|S(v)|)$ space, and hence the overall space required for T is $\sum_{v \in T} O(|S(v)|) = O(n \log n)$. Also, it can be verified that the construction time for T is $O(n \log^2 n)$.

For simplicity in analysis, let the time to query a partition tree to find canonical nodes be $O(f_{hs}(n))$. It is easy to see that $\mathcal{V}_{hs}(v) = O(f_{hs}(|S(v)|))$ for any $v \in \mathcal{V}_i$, and hence the time to query $T_{hs}(v)$ to find the point with largest a_1-feature value for q takes $O(f_{hs}(|S(v)|))$ time. The query algorithm outputs at most one point in each iteration, hence performs $k + 1$ iterations to output k points of $HSSky(S, q)$. (k iterations to output k points and one terminal iteration in which no point is output.) In the ith iteration, the range query on T with range Z_{i-1} take $O(\log n)$ time and gives $|\mathcal{V}_i| = O(\log n)$ canonical node. Therefore, the time required for each iteration is

$\sum_{v \in \mathcal{V}_i} O(f_{hs}(|S(v)|) = O(f_{hs}(n) \log n)$, which results in a overall query time of $O((1 + kf_{hs}(n)) \log n)$. By substituting the values for $f_{hs}(n)$, it can be verified that the query time of T for points in d-dimensional coordinate space is $O((1 + k) \log^2 n)$ for $d = 2, 3$ and $O((1 + k)n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n)$ for $d \geq 4$.

Second data structure. Now we show how to improve the query time by augmenting the partition tree suitably. Note that the query time of our previous data structure is dominated by the query times of the partition trees associated with the canonical nodes of T. (The algorithm queries $O(\log n)$ partition trees corresponding to the $O(\log n)$ canonical nodes for each skyline point.) Therefore, a key to improve the query time is to reduce the number of partition tree queries to find the next skyline point. An important observation towards this end is that all the partition trees are queried with the same query halfspace q, so the searches can be coordinated. Based on this observation, and inspired by the idea of fractional cascading [2, 78, 79], we present a way of augmenting the partition tree so that the number of queries on the partition tree is reduced to one per skyline point.

We first build a partition tree, T_{hs}, on the coordinate values of the points in S. We store the points of a leaf node in sorted order of their a_1-feature attribute value, breaking ties based on a_2-feature attribute value. For any node $v \in T_{hs}$, let $S(v)$ denote the set of points stored at the subtree rooted at v (or at v if v is a leaf node). Also, let $p_{max}(S(v))$ be the point in $S(v)$ with largest a_1-feature value (ties broken in favor of the point with highest a_2-feature value). At v, we store a linked-list, $L_{aug}(v)$, initialized with $p_{max}(S(v))$ to help us identify the skyline points. For any node $u \in T_{hs}$, if u is a child of v, we create a pointer from the the first entry of $L_{aug}(v)$ to the first entry of $L_{aug}(u)$.

Let $\alpha_1, \ldots, \alpha_{n'}$ be the sorted list of distinct a_2-feature attribute values of the points in S in increasing order, and let P_i be the subset of points of S with a_2-feature attribute value equal to α_i. For $i = 1, \ldots, n'$, we iteratively update the augmented lists as follows. Let $\mathcal{V}_i$ be the set of nodes in the union of the paths from the root of T_{hs} to the leaf nodes containing the points in P_i. For each $v \in \mathcal{V}_i$, we add an entry at the end of the list $L_{aug}(v)$ and store $p_{max}(S(v) \setminus \bigcup_{j=1}^{i} P_j)$. (If $S(v) \setminus \bigcup_{j=1}^{i} P_j = \emptyset$, we store $\emptyset$.) Also, for any node $u \in T_{hs}$, if u is a child of v, we create a pointer from the last entry of

$L_{aug}(v)$ to the last entry of $L_{aug}(u)$.

Observe that the list stored at the root contains n' elements, one corresponding to each value of α_i ($1 \leq i \leq n'$). Also, the tree structure formed by following the pointers from the ith list entry gives the partition tree augmented for the points with a_2-feature value in the range $[\alpha_i, \alpha_{n'}] = (\alpha_{i-1}, \infty)$. At the root, we also store an array containing $\alpha_1, \ldots, \alpha_{n'}$ and create pointers to the corresponding list element to easily access the augmentation values for any given α_i. This completes the construction of our data structure.

An example construction of the above data structure is shown in Figure 6.4. Figure 6.4a and Figure 6.4b shows the points in coordinate space and in feature space, respectively. The partition tree T_{hs} is shown in Figure 6.4c and the lists augmented at the nodes of the partition tree are shown in Figure 6.4d. Here $\alpha'_1, \ldots, \alpha'_{11}$ are distinct a_2-feature attribute values of the points in increasing order. For each node $v \in T_{hs}$, the first element of $L_{aug}(v)$ stores $p_{max}(S(v))$. (In this example $p_{max}(S(v_a)) = p_{16}$, $p_{max}(S(v_b)) = p_9$, $p_{max}(S(v_c)) = p_{12}$, etc.) The remaining list elements are computed as follows. First, we search T_{hs} and get $\mathcal{V}_1 = \{v_a, v_b, v_c, v_f, v_i\}$ as the set of nodes in the union of the paths from the root to the leaves containing the points in $P_1 = \{p_4, p_7\}$, i.e., points with a_2-feature attribute value equal to α'_1. For each $v \in \mathcal{V}_1$, we add an entry in $L_{aug}(v)$ and store $p_{max}(S(v) \setminus P_1)$, which in this case is p_{16} (resp., p_9, p_{12}, p_3, or p_8) for v_a (resp., v_b, v_c, v_f, or v_i). Next, for each $v \in \mathcal{V}_2 = \{v_a, v_c, v_j\}$ (i.e., the set of nodes in the path from the root to the leaves containing points in $P_2 = \{p_{12}\}$), we add an entry in $L_{aug}(v)$ and store $p_{max}(S(v) \setminus (P_1 \cup P_2))$, which in this case is p_{16} (resp., p_6 or p_{13}) for v_a (resp., v_c or v_j). We continue this for each α'_i, $i = 3, \ldots, 11$. Note that the $L_{aug}(v)$ entry that can be accessed by following the pointers from α_i stores the point of $S(v)$ with largest a_1-feature value (ties broken in favor of the point with highest a_2-feature value) such that a_2-feature value is at least α'_i.

Let $z = -\infty$ and $HSSky(S, q) = \emptyset$. To answer a Halfspace-Range-Skyline query for a given query halfspace q, we iterate on the following steps. On the ith iteration, we first query the array associated with the root of T_{hs} to locate the smallest entry larger than z and get the corresponding list element, e. Next, we query T_{hs} with the halfspace q and find a set, $\mathcal{V}_{hs}$, of nodes that cover the query halfspace range. Also, as we traverse down the nodes in T_{hs}, we follow the corresponding pointers from e and get

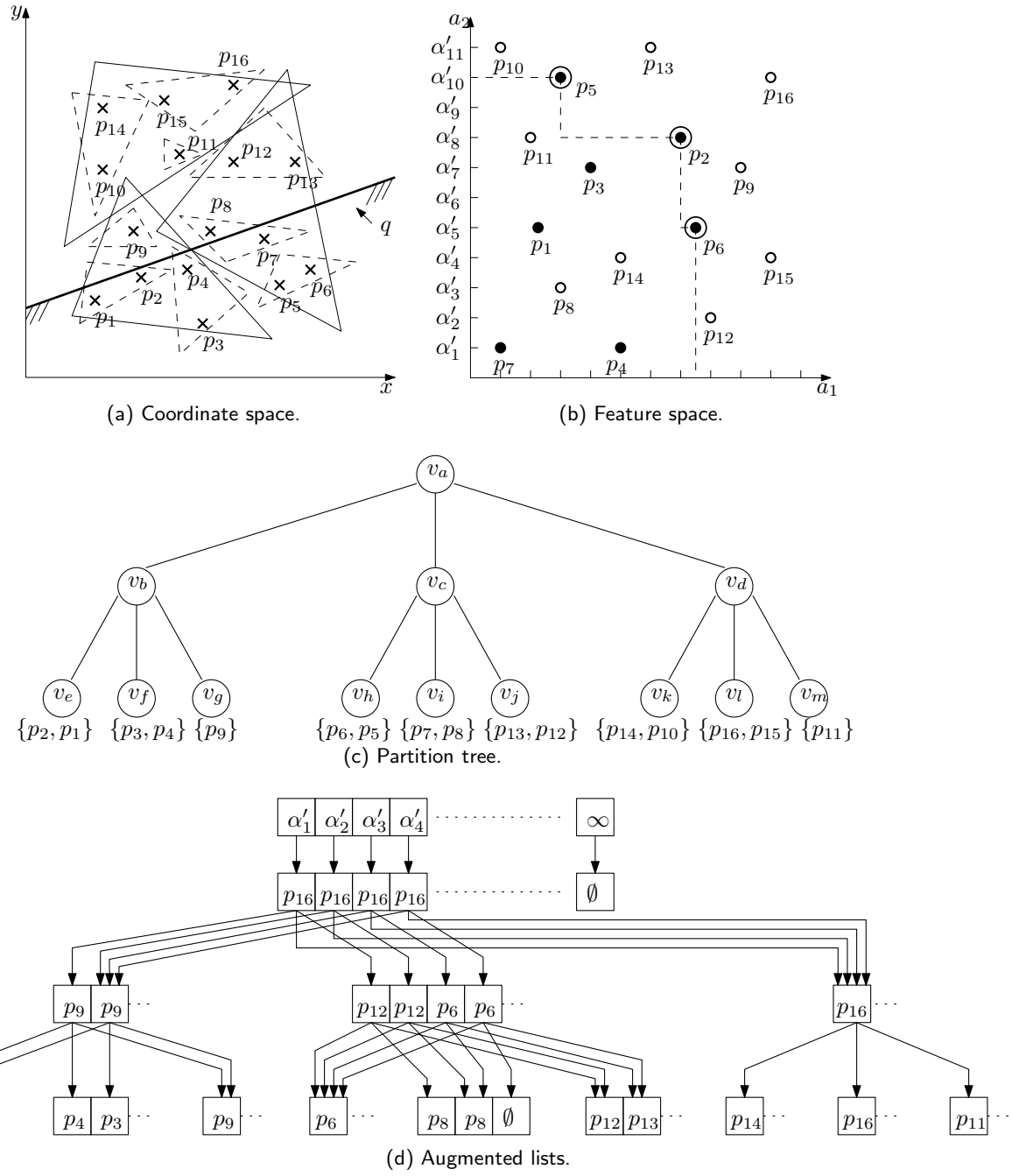


Figure 6.4: An example construction of our second data structure for the point set of Figure 6.1. (a) The partition of points in coordinate space for the partition tree. Here q is the query halfspace. (b) Points in the feature space. (c) The partition tree. (d) The lists augmented at each node of the partition tree. For simplicity in explanation, we have shown the augmented lists separate from the partition tree. Also, we have shown only a partial construction.

the augmented $p_{max}(\cdot)$ points with a_2-feature attribute value in the range (z, ∞) from the lists $L_{aug}(v)$ for each $v \in \mathcal{V}_{hs}$. (We check all points stored in v if v is a leaf node.) Now, as before, we compare these points and report the point, p, with largest a_1-feature value. We set z to be $a_2(p)$ and start the next iteration. We terminate the algorithm, if no point is reported in an iteration or if $|\mathcal{V}_{hs}| = 0$.

For example consider Figure 6.4. We answer a HALFSPACE-RANGE-SKYLINE query on the set S of points shown in Figure 6.4a for the query halfspace q (indicated by hatched area) as follows. First, we initialize $z = -\infty$. We search the array associated with the root of T_{hs} and get the smallest entry larger than $z = -\infty$, which in this case is α'_1. Next, we query T_{hs} with q and find $\mathcal{V}_{hs} = \{v_e, v_f, v_h, v_i\}$. By following the pointers from α'_1 while traversing the tree, we get $p_{max}(\cdot)$ points from $L_{aug}(v)$ for each $v \in \mathcal{V}_{hs}$, which in this case are p_2, p_4, p_6, and p_8 for v_e, v_f, v_h, and v_i, respectively. Clearly, p_6 is the point with largest a_1-feature value. We report p_6. This completes the first iteration.

For the second iteration, we set $z = a_2(p_6) = \alpha'_5$ and query the array associated with the root to get α'_6 as the smallest entry larger than z. Now, we query our data structure as before and get p_2, p_3, and p_5 as the $p_{max}(\cdot)$ points from $L_{aug}(\cdot)$ for v_e, v_f, and v_h, respectively. (The entry in $L_{aug}(v_i)$ reachable by following the pointer from α'_6 is empty, indicating that the points in v_i have a_2-feature attribute values less than α'_6.) Since p_2 is the point with largest a_1-feature value, we report p_2. Similarly, in the third iteration, we set $z = a_2(p_2) = \alpha'_8$ and report p_5. In the fourth iteration, we set $z = a_2(p_5) = \alpha'_{10}$. It can be verified that, in this iteration, we do not report any point. Hence, we terminate the algorithm.

The space required by T_{hs} depends on the space required by the partition tree and the total space required by all the lists. The partition tree takes $O(n)$ space. The total space required for storing the lists $L_{aug}(v)$ with each $v \in T_{hs}$ is as follows. The initial space required by the lists is the same as the partition tree, i.e., $O(n)$. Note that, the partition tree is an n^{ε}-ary tree with height (i.e., length of the path from root to any leaf node) $O(\log \log n)$. (Here, $\varepsilon > 0$ is a real.) Therefore, for any α_i, $|\mathcal{V}_i| = O(|P_i| \log \log n)$. For each $v \in \mathcal{V}_i$, we create one new entry in $L_{aug}(v)$ and n^{ε} pointers to children, resulting in a space increase of $O(|P_i| n^{\varepsilon} \log \log n)$ for each α_i. Therefore, the total space required for the augmented lists is $O(n) + \sum_{i=1}^{n'} O(|P_i| n^{\varepsilon} \log \log n) = O(n^{1+\varepsilon} \log \log n)$ resulting in $O(n^{1+\varepsilon} \log \log n)$ as the total space usage for our data structure T_{hs}. Also, it can be

verified that the total construction time of T_{hs} is also $O(n^{1+\varepsilon} \log\log n)$.

The query time for T_{hs} is as follows. The query algorithm outputs at most one point in each iteration, hence performs $k+1$ iterations to output k points of $HSSky(S, q)$. (One additional iteration for the termination condition.) For simplicity in analysis, let the time to query a partition tree to find the set of canonical nodes that cover a query halfspace range be $O(f_{hs}(n))$. For each iteration, the time required to find $\mathcal{V}_{hs}$, and also to traverse the lists is $O(f_{hs}(n))$. Also, it is easy to see that $\mathcal{V}_{hs} = O(f_{hs}(n))$, and hence the time required for each iteration is $O(f_{hs}(n))$. Therefore, the overall query time is $O((1+k)f_{hs}(n))$. By substituting the values for $f_{hs}(n)$, it can be verified that the query time of T_{hs} for points in d-dimensional coordinate space is $O((1+k)\log n)$ for $d = 2, 3$ and $O((1+k)n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n)$ for $d \geq 4$.

Theorem 6.1. *A* Halfspace-Range-Skyline *problem on a set of n point in d-dimensional coordinate space, where each point has $t = 2$ real-valued feature attributes, can be solved in (i) $O((1+k)\log^2 n)$ time (resp., $O((1+k)n^{1-1/\lfloor d/2 \rfloor} \log n \log^{O(1)} n)$ time) for $d = 2, 3$ (resp., $d \geq 4$) using a data structure of $O(n \log n)$ space that can be constructed in $O(n \log^2 n)$ time, or (ii) $O((1+k)\log n)$ time (resp., $O((1+k)n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n)$ time) for $d = 2, 3$ (resp., $d \geq 4$) using a data structure of $O(n^{1+\varepsilon} \log\log n)$ space that can be constructed in $O(n^{1+\varepsilon} \log\log n)$ time.*

6.5 Halfspace-Range-Skyline for points with $d = 2, 3$ and $t \geq 3$

In this section, we discuss our approach for the Halfspace-Range-Skyline query problem (Problem 6.2) for points with $t \geq 3$ feature attribute values in d-dimensional coordinate space ($d = 2, 3$). Note that, unlike in 2-dimensional feature space, the skyline points in t-dimensional feature space ($t \geq 3$) do not exhibit a total order. Therefore, it is not possible to use our previous method to compute $HSSky(S, q)$ for points with $t \geq 3$ feature attribute values. In the rest of this section we discuss an alternative method to solve this problem. The main idea behind our algorithm is to transform the points of S to a set of colored points in such a way that the solution to a certain generalized (i.e., colored) halfspace range non-intersection problem [75] yields the solution to the problem at hand. The generalized halfspace range non-intersection problem is defined

as follows: Preprocess a set O of n colored points in d-dimensional coordinate space so that, for any query halfspace q, we can efficiently report the distinct colors in O that are *not* intersected by q. Here q intersects a color c if and only if q contains at least one point of color c.

For any point $p_i \in S$, let S_i be the set of points of S that dominate p_i in feature space. A necessary and sufficient condition for p_i to be a point in $HSSky(S, q)$ is that the query halfspace q contains p_i but no point of S_i, i.e., $p_i \in q \cap S$ and $q \cap S_i = \emptyset$. Based on this, we propose the following preprocessing steps. We first build a partition tree, T_{hs}, in the coordinate space on the points of S. For any node $v \in T_{hs}$, let $S(v)$ be the set of points stored at the subtree rooted at v, and let $\bar{S}(v)$ be the points in the skyline of $S(v)$ (computed in feature space). For each point $p_i \in \bar{S}(v)$, we compute S_i. For each point $p \in S_i$, we create a d-dimensional colored point with color $c_i = i$ and with coordinate values the same as those of p. Let C_i be the set of points with color c_i, and let $S_c(v) = \bigcup_{p_i \in \bar{S}(v)} C_i$. At v we build a generalized halfspace range non-intersection query data structure, $D_c(v)$ on $S_c(v)$ [75].

To answer a Halfspace-Range-Skyline query for a given query halfspace q, we first query T_{hs} with the query halfspace q and find a set of nodes, $\mathcal{V}_{hs}$, that cover the query halfspace range. Next, for each $v \in \mathcal{V}_{hs}$, we query $D_c(v)$ with q and get a set $C_q(v)$ of colors such that there are no points of those colors in $q \cap S_c(v)$. Note that if a color $c_i \in C_q(v)$ then $q \cap C_i = \emptyset$, i.e., $q \cap S_i = \emptyset$, and hence p_i is a skyline point. Therefore, we report the points p_i corresponding to the colors c_i in $C_q(v)$ as points of $HSSky(S, q)$.

We argue that a point p_i is reported if and only if p_i is a Halfspace-Range-Skyline point. Suppose that p_i is reported. Thus, there is a node $v \in \mathcal{V}_{hs}$ such that $p_i \in S(v)$. Hence, $p_i \in q \cap S$. Also, c_i is found when $D_c(v)$ is queried with q, i.e., $q \cap C_i = \emptyset$, i.e., $q \cap S_i = \emptyset$. Therefore, p_i is a skyline point. On the other hand, suppose p_i is a skyline point. Thus, $p_i \in q \cap S$ and $q \cap S_i = \emptyset$. Since, $p_i \in q \cap S$, there exists a node $v \in \mathcal{V}_{hs}$ such that $p_i \in S(v)$. Additionally, $p_i \in \bar{S}(v)$ (otherwise, p_i is dominated by some point in $S(v)$), and hence $C_i \in S_c(v)$. Also, $q \cap S_i = \emptyset$, i.e., $q \cap C_i = \emptyset$ and hence c_i is found when $D_c(v)$ is queried with q. Therefore, p_i is reported.

The space and the query time of our approach depend on the space and query time of the generalized halfspace range non-intersection query data structure. It has been

shown in [75] that a generalized halfspace range non-intersection query on n points in d-dimensional space can be answered in $O(n \log n)$ space and $O(\log^2 n + k')$ query time for $d = 2$, and $O(n \log^2 n)$ (resp., $O(n^{2+\varepsilon})$) space and $O(n^{1/2+\varepsilon} + k')$ (resp., $O(\log^2 n + k')$) query time for $d = 3$. Here k' is the number of distinct colors reported.

For simplicity in analysis, let the space and the query time of the generalized halfspace range non-intersection query data structure on n points be $S(n)$ and $O(f(n) + k')$, respectively. Note that, for a node $v \in T_{hs}$ at level i, $0 \leq i \leq O(\log \log n)$ (the root is at level 0) the number of points stored in the subtree rooted at v is $|S(v)| = n^{1-i\varepsilon}$. Also, for each point $p_i \in S(v)$, the number of points of S that dominate p_i in feature space is at most n. Thus, at v, $|S_c(v)| = \sum_{p_i \in \bar{S}(v)} |C_i| = \sum_{p_i \in S(v)} |S_i| \leq n^{2-i\varepsilon}$, and hence the space required by $D_c(v)$ is $O(S(n^{2-i\varepsilon}))$. The number of nodes at level i is $O(n^{i\varepsilon})$. Therefore, the total space required by our data structure in $\sum_{i=0}^{\log \log n} O(S(n^{2-i\varepsilon}))O(n^{i\varepsilon})$. Also, it is easy to see that the query time of our approach is $\sum_{v \in \mathcal{V}_{hs}} O(f(n) + k_v) = O(f(n)|\mathcal{V}_{hs}|+k)$, where k_v is the number of distinct colors reported at v (which equals the number of skyline points reported at v) and $k = \sum_{v \in \mathcal{V}_{hs}} k_v$. By substituting the bounds for the number of canonical nodes in the partition tree and the generalized halfspace range non-intersection query, it can be verified that the bounds for the Halfspace-Range-Skyline problem are as listed in Theorem 6.2.

Theorem 6.2. *A* Halfspace-Range-Skyline *problem on a set of n point in d-dimensional coordinate space, where each point has $t \geq 3$ real-valued feature attributes, can be solved in $O(n^2 \log n \log \log n)$ space and $O(\log^3 n + k)$ query time for $d = 2$, and $O(n^2 \log^2 n \log \log n)$ (resp., $O(n^{4+\varepsilon} \log \log n)$) space and $O(n^{1/2+\varepsilon} \log n + k)$ (resp., $O(\log^3 n + k)$) query time for $d = 3$.*

6.6 The SkyCTRQ problem

In this section, we discuss our solution for the SkyCTRQ problem (Problem 6.1). We present efficient solutions for the fixed distance and the variable distance version of the SkyCTRQ problem by mapping these to suitable instances of the Halfspace-Range-Skyline query problem. We first discuss the fixed distance version of the problem.

For any $b_i \in B$, let the *Pareto-optimal tuples of* b_i, denoted by $Pareto(b_i)$, be defined as a subset of $\{b_i\} \times R$ with the following property. Let (b_i, r_j) and (b_i, r_l) be any two

tuples in $\{b_i\} \times R$ such that $d(b_i, r_l) \leq d(b_i, r_j)$. Then $(b_i, r_j) \in \mathit{Pareto}(b_i)$ if and only if $(b_i, r_l) \not\succ (b_i, r_j)$, i.e., $a(r_l) \leq a(r_j)$. (Note that our definition of Pareto-optimal tuples and the construction below are slightly different from those in Chapter 5 as our goal here is different.)

To compute $\mathit{Pareto}(b_i)$ for a b_i, we build a binary search tree T_i based on the feature attribute values of points in R and their distances from b_i, as follows. Let $r_j \in R$ be the point with largest feature attribute value (ties are resolved by favoring the point with highest distance to b_i.) At the root of T_i, we store (b_i, r_j). Let R' be the subset of R with distance to b_i at least $d(b_i, r_j)$ and attribute value less than $a(r_j)$. The left (resp., right) subtree is built recursively on the subset of points of R' (resp., $R \setminus R'$).

It is clear from the construction that the nodes of T_i have the following properties. Let v and v' be any two nodes of T_i, and let (b_i, r) and (b_i, r') be the tuples corresponding to v and v', respectively.

1. If v' is in the subtree rooted at v then $a(r') \leq a(r)$, i.e., $(b_i, r') \not\succ (b_i, r)$.

2. Node v' is in the subtree rooted at the left child of v if and only if $d(b_i, r) \leq d(b_i, r')$ and $a(r) > a(r')$, i.e., $(b_i, r) \succ (b_i, r')$.

We start from the root and traverse the tree as follows. At any node, we add the corresponding (blue, red) tuple to an initially empty set, $\mathit{Pareto}(b_i)$, and go to its right child. We stop the traversal if the search falls off the tree, i.e. right child of the node does not exist. We call the above-traversed path as *right chain*.

We argue that $\mathit{Pareto}(b_i)$ computed as above has the desired property. For any tuple $(b_i, r_j) \in \{b_i\} \times R$, let v_j be the corresponding node in T_i. Suppose $(b_i, r_j) \in \mathit{Pareto}(b_i)$. Clearly, v_j is a node in the right chain. Let $(b_i, r_l) \in \{b_i\} \times R$ be any tuples such that $d(b_i, r_l) \leq d(b_i, r_j)$ and let v_l be the corresponding nodes in T_i. If v_l is in the subtree rooted at v_j then $(b_i, r_l) \not\succ (b_i, r_j)$ (Property 1 of T_i). Otherwise, v_j is in the right subtree rooted at v_l, and hence, since $d(b_i, r_l) \leq d(b_i, r_j)$, $(b_i, r_l) \not\succ (b_i, r_j)$ (Property 2 to T_i). On the other hand suppose $(b_i, r_j) \notin \mathit{Pareto}(b_i)$. Let v_l be the last node of the right chain in the path from root to v_j and let $(b_i, r_l) \in \mathit{Pareto}(b_i)$ be the corresponding tuple. Clearly, v_j is in subtree rooted at the left child of v_l. Therefore, based on the previously-stated properties of T_i, $d(b_i, r_l) \leq d(b_i, r_j)$ and $(b_i, r_l) \succ (b_i, r_j)$.

For a $b_i \in B$, the number of points in $Pareto(b_i)$ is same as the length of the right chain of the binary search tree T_i, which is bounded by the height of T_i. In the average case, i.e. when the attribute values are assigned uniformly at random, T_i is a *random binary search tree*, as noted in [72]. Therefore, even though in the worst case, the height of T_i is $O(|R|)$, in the average case the height of T_i is $O(\log n)$ [73]. Therefore, the expected size of $Pareto(b_i)$ is $O(\log n)$.

Lemma 6.1. *For any $b_i \in B$, let $Pareto(b_i)$ be the set of tuples as computed above. Let q be the query point and δ be the (fixed) query distance. If (b_i, r_j) is a tuple in the answer to the* SkyCTRQ *query for q and δ then $(b_i, r_j) \in Pareto(b_i)$.*

Proof. For a contradiction, let us assume that $(b_i, r_j) \notin Pareto(b_i)$. Then, according to the definition of $Pareto(b_i)$, there is a tuple $(b_i, r_l) \in Pareto(b_i)$ such that $d(b_i, r_l) \leq d(b_i, r_j)$, i.e., $d(q, b_i) + d(b_i, r_l) \leq d(q, b_i) + d(b_i, r_j) \leq \delta$. Also, $(b_i, r_l) \succ (b_i, r_j)$, and hence (b_i, r_j) cannot be a tuple in the answer set of SkyCTRQ for q and δ. □

In other words, if a tuple (b_i, r_l) is not in $Pareto(b_i)$ then (b_i, r_l) cannot be in the answer set of the SkyCTRQ query for any q and δ. Note that whether a tuple of $Pareto(b_i)$ is in the answer set of the SkyCTRQ query or not depends on q and δ. Based on this, we first compute $Pareto(b_i)$, for each $b_i \in B$, as described above. Let $Pareto(B) = \bigcup_{b_i \in B} Pareto(b_i)$. Now, we apply the following transformation to map the SkyCTRQ problem to an instance of the Halfspace-Range-Skyline query problem (Problem 6.2) on a set S of points in 3-dimensional coordinate space, where each point has $t = 2$ feature attribute values.

1. Tuple (b_i, r_j) is mapped to a point, p_{ij}, in 3-dimensional coordinate space with coordinate values $(x(b_i), y(b_i), z_{ij})$ and feature attribute values $(a(b_i), a(r_j))$, where $x(b_i)$ and $y(b_i)$ are the coordinate values of b_i and $z_{ij} = x(b_i)^2 + y(b_i)^2 + 2\delta d(b_i, r_j) - (d(b_i, r_j))^2$.

2. Query point q with coordinate values $(x(q), y(q))$ is mapped to a halfspace q' in 3-dimensional coordinate space defined by the equation $\beta_1 x + \beta_2 y + \beta_3 z \leq c$, where $\beta_1 = -2x(q)$, $\beta_2 = -2y(q)$, $\beta_3 = 1$, and $c = \delta^2 - x(q)^2 - y(q)^2$.

Lemma 6.2. *Consider the transformation given above. Then, for any* $(b_i, r_j) \in Pareto(B)$, (b_i, r_j) *is a tuple in the answer set of the fixed distance* SkyCTRQ *query for* q *and* δ *if and only if* p_{ij} *is in* $HSSky(S, q')$.

Proof. Note that the feature attribute values of p_{ij} are the same as that of the tuple (b_i, r_j). Therefore, to prove this lemma, it is sufficient to prove that $d(q, b_i)+d(b_i, r_j) \leq \delta$ if and only if p_{ij} is in the query halfspace q'.

We have

$$
\begin{aligned}
& d(q, b_i) + d(b_i, r_j) \leq \delta \\
\Leftrightarrow \quad & \sqrt{(x(q) - x(b_i))^2 + (y(q) - y(b_i))^2} \leq \delta - d(b_i, r_j) \\
\Leftrightarrow \quad & x(q)^2 + y(q)^2 + x(b_i)^2 + y(b_i)^2 - 2x(q)x(b_i) - 2y(q)y(b_i) \\
& \qquad\qquad \leq \delta^2 + (d(b_i, r_j))^2 - 2\delta d(b_i, r_j) \\
\Leftrightarrow \quad & (-2x(q))x(b_i) + (-2y(q))y(b_i) + (1)(x(b_i)^2 + y(b_i)^2 + 2\delta d(b_i, r_j) - (d(b_i, r_j))^2) \\
& \qquad\qquad \leq \delta^2 - x(q)^2 - y(q)^2 \\
\Leftrightarrow \quad & \beta_1 x(b_i) + \beta_2 y(b_i) + \beta_3 z_{ij} \leq c \\
\Leftrightarrow \quad & p_{ij} \text{ is on or below } q'.
\end{aligned}
$$

□

Based on Lemma 6.2, we use the solution for the Halfspace-Range-Skyline query from Section 6.4 on the points in transformed space to get the solution for the fixed distance version of the SkyCTRQ problem.

Theorem 6.3. *A fixed-distance* SkyCTRQ *problem on sets of* n *blue and red points in the plane, where each point has one additional real-valued feature attribute, can be solved in* (i) $O((1+k)\log^2 n)$ *time and* $O(n \log^2 n)$ *expected space, or* (ii) $O((1+k)\log n)$ *time and* $O((n \log n)^{1+\varepsilon} \log\log n)$ *expected space. Here* k *is the output size.*

Handling variable distance. Now we present our approach for the variable distance SkyCTRQ problem. As before, we transform the problem at hand to a Halfspace-Range-Skyline query problem in a different space. Note that Lemma 6.1 and the approach to compute $Pareto(b_i)$, for any $b_i \in B$, continues to hold even if δ is not known beforehand. Therefore, as before, we first compute the set $Pareto(B)$ of tuples.

Next, we apply the following transformation to map the variable distance SkyCTRQ problem to an instance of Halfspace-Range-Skyline query problem on points in 4-dimensional coordinate space, where each point has $t = 2$ feature attribute values.

1. Tuple (b_i, r_j) is mapped to a point, p_{ij}, in 4-dimensional coordinate space with coordinate values $(x(b_i), y(b_i), z_{ij}, w_{ij})$ and feature attribute values $(a(b_i), a(r_j))$, where $x(b_i)$ and $y(b_i)$ are the coordinate values of b_i, $z_{ij} = x(b_i)^2 + y(b_i)^2 - (d(b_i, r_j))^2$, and $w_{ij} = d(b_i, r_j)$.

2. Query point q with coordinate values $(x(q), y(q))$ is mapped to a halfspace q' in 4-dimensional coordinate space defined by the equation $\beta_1 x + \beta_2 y + \beta_3 z + \beta_4 w \leq c$, where $\beta_1 = -2x(q)$, $\beta_2 = -2y(q)$, $\beta_3 = 1$, $\beta_4 = 2\delta$, and $c = \delta^2 - x(q)^2 - y(q)^2$.

Lemma 6.3. *Consider the transformation mentioned above. Then, for any $(b_i, r_j) \in Pareto(B)$, (b_i, r_j) is tuple in the answer set of the variable distance* SkyCTRQ *query for q and δ if and only if p_{ij} is a* Halfspace-Range-Skyline *point for query halfspace q'.*

The proof of the above lemma mirrors the proof of Lemma 6.2, and is hence omitted. Based on Lemma 6.3, we use the solution for the Halfspace-Range-Skyline query problem from Section 6.4 on the points in the transformed space to get the solution for the variable distance version of the SkyCTRQ problem.

Theorem 6.4. *A variable-distance* SkyCTRQ *problem on sets of n blue and red points in the plane, where each point has one additional real-valued feature attributes, can be solved in (i) $O((1+k)\log n\sqrt{n}\log^{O(1)} n)$ time and $O(n\log^2 n)$ expected space, or (ii) $O((1+k)\sqrt{n}\log^{O(1)} n)$ time and $O((n\log n)^{1+\varepsilon}\log\log n)$ expected space. Here k is the output size.*

Chapter 7

Conclusions and Future Work

In this dissertation, we have studied a new class of geometric search problems, called *geometric search on point-tuples*, where the goal is to report point-tuples that satisfy query constraints. Specifically, we have studied the nearest neighbor search and range search problems in the context of point-tuples.

In Chapter 2, we have presented our solution for the point-tuple version of the nearest neighbor search query problem, called the *nearest neighbor color tuple query* (NNCTQ) problem. Given a query point q, the goal is to report a point-tuples of the given ordering of the colors such that the total distance from q through the points of the tuple is minimum. We have given a two-step approach to solve this problem. First, we have shown how to efficiently reduce the given problem to a weighted nearest neighbor problem. Next, we have given three linear-space algorithms, with different query performance characteristics, to solve the weighted nearest neighbor problem efficiently.

In Chapter 3, we have formulated the point-tuple version of the range search query problem. Given q and a distance δ, the goal is to report all point-tuples of the given ordering of the colors, such that the total distance from q through the sequence is no more than δ. We call this the *color tuple range query* (CTRQ) problem. We have given efficient solutions for the fixed distance and the variable distance versions of this problem, where δ is known beforehand or known only at query time, respectively. We have also shown a technique to tune the query algorithm to the (unknown) number of output tuples to achieve a space-query time tradeoff.

We have also studied several different versions of the CTRQ problem when the

input points have real-valued feature attribute values (in addition to coordinate values) and have devised different techniques to incorporate informations about the feature attributes during tuple identification.

In Chapter 4, we have studied the CTRQ problem with constraints on the feature attributes. Given q, δ, and a set of range constraints on feature attributes, $\mathcal{Z}$, the goal is to report all point-tuples of the given ordering of colors such that the tuples satisfies the distance constraint and the points satisfy the feature constraints. For this problem, we have shown that a certain minimum amount of storage is unavoidable to achieve a fast query time. This is via a reduction from the well-known Set Intersection problem [71]. We have also given efficient solutions that uses no more than this amount of storage for the fixed distance and the variable distance versions of the problem.

In Chapter 5, we have studied a version of the CTRQ problem where, for a given q, δ, and a threshold constraint τ, the goal is to report all point-tuples of the given ordering of colors such that each tuple satisfies the distance constraint and has the score no less than τ. Here, the score of a tuple is computed via some aggregation function (e.g., sum, product, etc.) on the feature attribute values of the points in the tuple. We have shown efficient solutions for four versions of the problem, where δ and τ are fixed or variable. An obvious open question here is to improve upon the algorithms. Obtaining a non-trivial lower bound for this problem is also an important open problem. Another interesting direction for future research is to allow users to specify the scoring function for the tuples at query time. This is quite natural since preferences of different users can be modeled as different scoring functions.

Finally, in Chapter 6, we have extended the CTRQ problem to report the *skyline* of the tuples that satisfy the distance constraint (δ). This is equivalent to a point-tuple version of the range-skyline query problem. Here, a tuple satisfying the distance constraint is in the skyline if and only if it contains at least one point that is not dominated by the points of the corresponding color in the tuples that satisfies the distance constraint. Our approach involves reducing the problem at hand to a Halfspace-Range-Skyline query problem, where the goal is to report the skyline of the points (in feature space) that lie is the query halfspace (in coordinate space). We have given efficient algorithms for various instances of the Halfspace-Range-Skyline query problem.

An interesting future research direction here is to solve the HALFSPACE-RANGE-SKYLINE problem for points with more than 2 feature attributes in d-dimensional coordinate space ($d \geq 4$). This also seems to be a key to solve the point-tuple version of the range-skyline query problem for more than two colors. Another important problem is to obtain a non-trivial lower bound for the HALFSPACE-RANGE-SKYLINE query problem, even for points with two feature attributes. While it seems difficult to obtain a linear space data structure with $O(\log n + k)$ query time for this problem, some of the logarithmic factors could perhaps be removed.

References

[1] Michael Ian Shamos. Computational geometry. *Ph. D. Dissertation, Yale University*, 1978.

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. Springer, 2008.

[3] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1988.

[4] Alexandr Andoni. Nearest neighbor search: the old, the new, and the impossible. *Ph.D. Dissertation, Massachusetts Institute of Technology*, 2009.

[5] Franz Aurenhammer. Voronoi Diagrams–a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[6] Kenneth L. Clarkson. Nearest-neighbor searching and metric space dimensions. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59, 2006.

[7] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[8] Pankaj K. Agarwal. Range searching. In *CRC Handbook of Discrete and Computational Geometry*. CRC Press, Inc., 2004.

[9] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. *Contemporary Mathematics*, 223:1–56, 1999.

[10] Akash Agrawal, Saladi Rahul, Yuan Li, and Ravi Janardan. Range search on tuples of points. *Journal of Discrete Algorithms*, 30:1–12, 2015.

[11] Stephan Börzsöny, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430. IEEE, 2001.

[12] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[13] Christos H. Papadimitriou and Mihalis Yannakakis. Multiobjective query optimization. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 52–59, 2001.

[14] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In *VLDB*, volume 1, pages 301–310, 2001.

[15] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Jun Xu. Randomized multi-pass streaming skyline algorithms. *VLDB*, 2(1):85–96, 2009.

[16] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 467–478, 2003.

[17] Shiming Zhang, Nikos Mamoulis, and David W Cheung. Scalable skyline computation using object-based space partitioning. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 483–494, 2009.

[18] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 275–286. VLDB Endowment, 2002.

[19] Cheng Sheng and Yufei Tao. Worst-case I/O-efficient skyline algorithms. *ACM Transactions on Database Systems (TODS)*, 37(4):26, 2012.

[20] Casper Kejlberg-Rasmussen, Yufei Tao, Konstantinos Tsakalidis, Kostas Tsichlas, and Jeonghun Yoon. I/O-efficient planar range skyline and attrition priority queues.

In *Proceedings of the 32nd ACM Symposium on Principles of Database Systems*, pages 103–114, 2013.

[21] Saladi Rahul and Ravi Janardan. Algorithms for range-skyline queries. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 526–529. ACM, 2012.

[22] Andrew Moore. A tutorial on kd-trees. *Technical report, University of Cambridge Computer Laboratory*, 1991.

[23] King Lum Cheung and Ada Wai-Chee Fu. Enhanced nearest neighbour search on the R-tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.

[24] Andreas Henrich. A distance-scan algorithm for spatial access structures. In *Proceedings of the Second ACM Workshop on Geographic Information Systems*, pages 136–143, 1994.

[25] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. *ACM SIGMOD Record*, 24(2):71–79, 1995.

[26] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems*, 30(2):529–576, 2005.

[27] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[28] Der-Tsai Lee. On k-nearest neighbor Voronoi Diagrams in the plane. *IEEE Transactions on Computers*, 100(6):478–487, 1982.

[29] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[30] Ahmed M. Aly, Walid G. Aref, and Mourad Ouzzani. Spatial queries with two kNN predicates. *Proceedings of the VLDB Endowment*, 5(11):1100–1111, 2012.

[31] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based k-nearest neighbor search for spatial network databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*, volume 30, pages 840–851, 2004.

[32] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proceedings of the 29th International Conference on Very Large Data Bases*, volume 29, pages 802–813, 2003.

[33] Liang Zhu, Yinan Jing, Weiwei Sun, Dingding Mao, and Peng Liu. Voronoi-based aggregate nearest neighbor query processing in road networks. In *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 518–521, 2010.

[34] Herbert Edelsbrunner. *Algorithms in combinatorial geometry*, volume 10. Springer, 1987.

[35] Peyman Afshani and Timothy M. Chan. Optimal halfspace range reporting in three dimensions. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 180–186. Society for Industrial and Applied Mathematics, 2009.

[36] Fabrizio d'Amore, Paolo Giulio Franciosa, Roberto Giaccio, and Maurizio Talamo. Maintaining maxima under boundary updates. In *Algorithms and Complexity*, pages 100–109. Springer, 1997.

[37] Greg N. Frederickson and Susan Rodger. A new approach to the dynamic maintenance of maximal points in a plane. *Discrete & Computational Geometry*, 5(4):365–374, 1990.

[38] Ravi Janardan. On the dynamic maintenance of maximal points in the plane. *Information processing letters*, 40(2):59–64, 1991.

[39] Sanjiv Kapoor. Dynamic maintenance of maxima of 2-d point sets. *SIAM Journal on Computing*, 29(6):1858–1877, 2000.

[40] Mark H. Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of computer and System Sciences*, 23(2):166–204, 1981.

[41] Gerth Stølting Brodal and Konstantinos Tsakalidis. Dynamic planar range maxima queries. In *Automata, Languages and Programming*, pages 256–267. Springer, 2011.

[42] Anil Kishore Kalavagattu, Ananda Swarup Das, Kishore Kothapalli, and Kannan Srinathan. On finding skyline points for range queries in plane. In *Proceedings of the 23rd Annual Canadian Conference on Computational Geometry*, 2011.

[43] Ananda Swarup Das, Prosenjit Gupta, Anil Kishore Kalavagattu, Jatin Agarwal, Kannan Srinathan, and Kishore Kothapalli. Range aggregate maximal points in the plane. In *WALCOM: Algorithms and Computation*, pages 52–63. Springer, 2012.

[44] Nadeem Moidu, Jatin Agarwal, Sankalp Khare, Kishore Kothapalli, and Kannan Srinathan. On generalized planar skyline and convex hull range queries. In *Algorithms and Computation*, pages 34–43. Springer, 2014.

[45] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. The optimal sequenced route query. *The VLDB journal*, 17(4):765–787, 2008.

[46] Xiaobin Ma, Shashi Shekhar, Hui Xiong, and Pusheng Zhang. Exploiting a page-level upper bound for multi-type nearest neighbor queries. In *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems*, pages 179–186, 2006.

[47] Baihua Zheng, Ken CK Lee, and Wang-Chien Lee. Transitive nearest neighbor search in mobile environments. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, volume 1, pages 14–21. IEEE, 2006.

[48] Xiao Zhang, Wang-Chien Lee, Prasenjit Mitra, and Baihua Zheng. Processing transitive nearest-neighbor queries in multi-channel access environments. In *Proceedings of the 11th ACM International Conference on Extending Database Technology: Advances in Database Technology*, pages 452–463, 2008.

[49] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *Advances in Spatial and Temporal Databases*, pages 273–290. Springer, 2005.

[50] Yaron Kanza, Eliyahu Safra, Yehoshua Sagiv, and Yerach Doytsher. Heuristic algorithms for route-search queries over geographical data. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 11, 2008.

[51] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. The partial sequenced route query with traveling rules in road networks. *Geoinformatica*, 15(3):541–569, 2011.

[52] Xiaobin Ma, Shashi Shekhar, and Hui Xiong. Multi-type nearest neighbor queries in road networks with time window constraints. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 484–487, 2009.

[53] Dongxiang Zhang, Chee-Yong Chan, and Kian-Lee Tan. Nearest group queries. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 7, 2013.

[54] Roy Levin and Yaron Kanza. Interactive traffic-aware route search on smartphones. In *Proceedings of the First ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems*, pages 1–8, 2012.

[55] Roy Levin and Yaron Kanza. TARS: traffic-aware route search. *GeoInformatica*, pages 1–40, 2013.

[56] Xin Cao, Lisi Chen, Gao Cong, Jihong Guan, Nhan-Tue Phan, and Xiaokui Xiao. KORS: Keyword-aware Optimal Route Search System. In *IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1340–1343. IEEE, 2013.

[57] Tanzima Hashem, Tahrima Hashem, Mohammed Eunus Ali, and Lars Kulik. Group trip planning queries in spatial databases. In *Advances in Spatial and Temporal Databases*, pages 259–276. Springer, 2013.

[58] Yaron Kanza, Eliyahu Safra, and Yehoshua Sagiv. Route search over probabilistic geospatial data. In *Advances in Spatial and Temporal Databases*, pages 153–170. Springer, 2009.

[59] Jing Li, Yin David Yang, and Nikos Mamoulis. Optimal route queries with arbitrary order constraints. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1097–1110, 2013.

[60] Xiaobin Ma, Chengyang Zhang, Shashi Shekhar, Yan Huang, and Hui Xiong. On multi-type reverse nearest neighbor search. *Data & Knowledge Engineering*, 70(11):955–983, 2011.

[61] Harald Rosenberger. Order-k voronoi diagrams of sites with additive weights in the plane. *Algorithmica*, 6(1-6):490–521, 1991.

[62] Peter F. Ash and Ethan D. Bolker. Generalized Dirichlet tessellations. *Geometriae Dedicata*, 20(2):209–243, 1986.

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

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

[65] Der-Tsai Lee and Franco P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM Journal on Computing*, 6(3):594–606, 1977.

[66] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[67] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, 1984.

[68] Gísli R Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[69] Apostolos Papadopoulos and Yannis Manolopoulos. Performance of nearest neighbor queries in R-trees. *Proceedings of the 6th International Conference on Database Theory*, pages 394–408, 1997.

[70] Bernard Chazelle, Richard Cole, Franco P. Preparata, and C. Yap. New upper bounds for neighbor searching. *Information and control*, 68(1):105–124, 1986.

[71] Pooya Davoodi, Michiel Smid, and Freek Van Walderveen. Two-dimensional range diameter queries. In *LATIN 2012: Theoretical Informatics*, pages 219–230. Springer, 2012.

[72] Subhash Suri and Kevin Verbeek. On the most likely Voronoi Diagram and nearest neighbor searching. In *Algorithms and Computation*, pages 338–350. Springer, 2014.

[73] Bruce Reed. The height of a random binary search tree. *Journal of the ACM (JACM)*, 50(3):306–332, 2003.

[74] Saladi Rahul and Ravi Janardan. A general technique for top-geometric intersection query problems. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):2859–2871, 2014.

[75] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Efficient non-intersection queries on aggregated geometric data. In *Computing and Combinatorics*, pages 544–553. Springer, 2005.

[76] Dan E. Willard. Polygon retrieval. *SIAM Journal on Computing*, 11(1):149–165, 1982.

[77] Timothy M. Chan. Optimal partition trees. *Discrete & Computational Geometry*, 47(4):661–690, 2012.

[78] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[79] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1(1-4):163–191, 1986.