5-2018

Geometric Algorithms for Intervals and Related Problems

Shimin Li
Utah State University

GEOMETRIC ALGORITHMS FOR INTERVALS AND RELATED PROBLEMS

by

Shimin Li

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

of

DOCTOR OF PHILOSOPHY

in

Computer Science

Approved:

Haitao Wang, Ph.D. Major Professor	David Brown, Ph.D. Committee Member
Curtis Dyreson, Ph.D. Committee Member	Amanda Lee Hughes, Ph.D. Committee Member
Minghui Jiang, Ph.D. Committee Member	Mark R. McLellan, Ph.D. Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2018

ABSTRACT

Geometric Algorithms for Intervals and Related Problems

by

Shimin Li, Doctor of Philosophy

Utah State University, 2018

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

In this dissertation, we study several problems related to intervals and develop efficient algorithms for them. Interval problems have many applications in reality because many objects, values, and ranges are intervals in nature, such as time intervals, distances, line segments, probabilities, etc. Problems on intervals are gaining attention also because intervals are among the most basic geometric objects, and for the same reason, computational geometry techniques find useful for attacking these problems. Specifically, the problems we study in this dissertation includes the following: balanced splitting on weighted intervals, minimizing the movements of spreading points, dispersing points on intervals, multiple barrier coverage, and separating overlapped intervals on a line. We develop efficient algorithms for these problems and our results are either first known solutions or improve the previous work.

In the problem of balanced splitting on weighted intervals, we are given a set of n intervals with non-negative weights on a line and an integer $k \geq 1$. The goal is to find k points to partition the line into $k + 1$ segments, such that the maximum sum of the interval weights in these segments is minimized. We give an algorithm that solves the problem in $O(n \log n)$ time. Our second problem is on minimizing the movements of spreading points. In this problem, we are given a set of points on a line and we want to spread the points on the line so that the minimum pairwise distance of all points is no smaller than a given value δ. The objective is to minimize the maximum moving distance of all points. We solve the problem in $O(n)$ time. We also solve the cycle version of the problem in linear time. For the third problem, we are given a set of n

non-overlapping intervals on a line and we want to place a point on each interval so that the minimum pairwise distance of all points are maximized. We present an $O(n)$ time algorithm for the problem. We also solve its cycle version in $O(n)$ time. The fourth problem is on multiple barrier coverage, where we are given n sensors in the plane and m barriers (represented by intervals) on a line. The goal is to move the sensors onto the line to cover all the barriers such that the maximum moving distance of all sensors is minimized. Our algorithm for the problem runs in $O(n^2 \log n \log\log n + nm \log m)$ time. In a special case where the sensors are all initially on the line, our algorithm runs in $O((n+m)\log(n+m))$ time. Finally, for the problem of separating overlapped intervals, we have a set of n intervals (possibly overlapped) on a line and we want to move them along the line so that no two intervals properly intersect. The objective is to minimize the maximum moving distance of all intervals. We propose an $O(n \log n)$ time algorithm for the problem.

The algorithms and techniques developed in this dissertation are quite basic and fundamental, so they might be useful for solving other related problems on intervals as well.

(164 pages)

PUBLIC ABSTRACT

Geometric Algorithms for Intervals and Related Problems

Shimin Li

In this dissertation, we study several problems related to intervals and develop efficient algorithms for them. Interval problems have many applications in reality because many objects, values, and ranges are intervals in nature, such as time intervals, distances, line segments, probabilities, etc. Problems on intervals are gaining attention also because intervals are among the most basic geometric objects, and for the same reason, computational geometry techniques find useful for attacking these problems. Specifically, the problems we study in this dissertation includes the following: balanced splitting on weighted intervals, minimizing the movements of spreading points, dispersing points on intervals, multiple barrier coverage, and separating overlapped intervals on a line. We develop efficient algorithms for these problems and our results are either first known solutions or improve the previous work.

In the problem of balanced splitting on weighted intervals, we are given a set of n intervals with non-negative weights on a line and an integer $k \geq 1$. The goal is to find k points to partition the line into $k + 1$ segments, such that the maximum sum of the interval weights in these segments is minimized. We give an algorithm that solves the problem in $O(n \log n)$ time. Our second problem is on minimizing the movements of spreading points. In this problem, we are given a set of points on a line and we want to spread the points on the line so that the minimum pairwise distance of all points is no smaller than a given value δ. The objective is to minimize the maximum moving distance of all points. We solve the problem in $O(n)$ time. We also solve the cycle version of the problem in linear time. For the third problem, we are given a set of n non-overlapping intervals on a line and we want to place a point on each interval so that the minimum pairwise distance of all points are maximized. We present an $O(n)$ time algorithm for the problem. We also solve its cycle version in $O(n)$ time. The fourth problem is on multiple barrier coverage, where we are given n sensors in the plane and

m barriers (represented by intervals) on a line. The goal is to move the sensors onto the line to cover all the barriers such that the maximum moving distance of all sensors is minimized. Our algorithm for the problem runs in $O(n^2 \log n \log\log n + nm \log m)$ time. In a special case where the sensors are all initially on the line, our algorithm runs in $O((n+m)\log(n+m))$ time. Finally, for the problem of separating overlapped intervals, we have a set of n intervals (possibly overlapped) on a line and we want to move them along the line so that no two intervals properly intersect. The objective is to minimize the maximum moving distance of all intervals. We propose an $O(n \log n)$ time algorithm for the problem.

The algorithms and techniques developed in this dissertation are quite basic and fundamental, so they might be useful for solving other related problems on intervals as well.

ACKNOWLEDGMENTS

This work would not have been done without the support and encouragement from individuals in the past four years.

I would like to express my deepest appreciation to my advisor Dr. Haitao Wang, who has the attitude and the substance of a genius, for his continuous support of my research and scholarship. His advice and guidance convincingly conveyed a spirit of adventure in regard to my research in our weekly meetings in these years. Without the persistent help from Dr. Haitao Wang, this dissertation would not have been possible.

Besides my advisor, I would like to thank the other members in my dissertation committee: Dr. David Brown, Dr. Curtis Dyreson, Dr. Amanda Lee Hughes, and Dr. Minghui Jiang for their insightful comments and constructive suggestions. Furthermore, their questions also encouraged me to widen my research from various perspectives. I also thank Ms. Jingru Zhang for inspiring technical discussions during these years. In addition, my sincere thanks go to the staff in Department of Computer Science for their administrative support. I really appreciate their help and support for my study.

I wish to express my full thanks to my wife and my parents. Without their love, support, and understanding, I could not have gone through the doctoral program overseas in four years. They are forever the source of my joy and happiness, and I have been appreciating that greatly.

Last but not least, I would like to thank my labmates and my friends. I shall never forget our delightful moments in my life together with them.

This work was sponsored in part by the National Science Foundation through Grant CCF-1317143.

Shimin Li

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In this dissertation, we study several problems related to intervals and develop efficient algorithms for them. Interval problems have many applications in reality because many objects, values, and ranges are intervals in nature, such as time intervals, distances, line segments, probabilities, etc. Problems on intervals are gaining attention also because intervals are among the most basic geometric objects, and for the same reason, computational geometry techniques find useful for attacking these problems.

In the rest of this chapter, we first briefly introduce the field of computational geometry and the topic of intervals, and we then present an overview on the problems we study in this dissertation. Finally, we will give an outline of the dissertation.

1.1 Computational Geometry

Computational geometry is a branch of computer science focusing on algorithm design, analysis, and implementation for solving geometric problems. The observations on the properties of the geometric objects in those problems play a critical role in the process of developing algorithms. The active research areas in computational geometry include both purely theoretical problems and problems arose from applications in the real world. Some important applications of computational geometry include computer graphics, geographic information systems, computer-aided engineering, computer vision, robotics, data analysis, facility location, and others. Refer to [1–4] for several great books on computational geometry.

1.2 Problems on Intervals

Certain objects and values in real applications can be treated as geometric intervals, such as time intervals, line segments, distances, probabilities, etc. Therefore, computational geometry techniques may be used to solve problems on intervals. Interval problems are in general very basic and the topic has been studied extensively in computational geometry and other fields. Algorithms for interval problems have many important applications, such as scheduling [5–11], and mobile sensor barrier coverage [12–17].

1.3 An Overview of Our Problems

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

1.3.1 Balanced Splitting on Weighted Intervals

This problem is motivated from load balancing in temporal and multi-version database systems. The problem can be formulated as follows. Let $\mathcal{I}$ be a set of n intervals on a line L, where each interval has a non-negative weight. For any given integer $k \geq 1$, we want to find k points on L to partition L into $k+1$ segments, such that the maximum cost of these segments is minimized, where the *cost* of each segment s is defined to be the sum of the weights of the intervals in $\mathcal{I}$ that intersect s. Previously, an $O(n \log n)$ time algorithm was given for a special case where the weights of all intervals are the same. We present an $O(n \log n)$ time algorithm for the general case where the intervals may have different weights.

Our results on this problem have been published in a journal [18]. Refer to Chapter 2 for the details.

1.3.2 Minimizing the Movements of Spreading Points

Given a set P of n points sorted on a line L and a distance value $\delta > 0$, the problem is to move the points of P along L such that the distance of any two points of P is at least δ and the maximum movement of all points of P is minimized. Using the greedy strategy, we present an $O(n)$ time algorithm for this problem. Further, we extend our algorithm to solve (in $O(n)$ time) the cycle version of the problem where all points of P are on a cycle C. Previously, only weakly polynomial-time algorithms were known for these problems based on linear programming (of n variables and $\Theta(n)$ constraints). In addition, we present a linear-time algorithm for another similar facility-location moving problem, which also improves the previous work.

Our results on this problem have been published in a conference [19]. Refer to Chapter 3 for the details.

1.3.3 Dispersing Points on Intervals

In certain applications, the movements of the points may be restricted. We consider a variation of the previous problem by adding some constraints on the movement of points. Given n pairwise disjoint intervals sorted on a line, we want to find a point in each interval such that the minimum pairwise distance of these points is maximized. We present a linear time algorithm for the problem. Further, we also solve in linear time the cycle version of the problem where the intervals are given on a cycle.

Our results on this problem have been published in a conference [20] and a journal [21]. Refer to Chapter 4 for the details.

1.3.4 Multiple Barrier Coverage

This problem is motivated from mobile sensor barrier coverage in wireless sensor networks. Given a set B of m line segments (called "barriers") on a horizontal line L and another set S of n horizontal line segments of the same length in the plane, we want to move all segments of S to L so that their union covers all barriers and the maximum movement of all segments of S is minimized. Previously, an $O(n^3 \log n)$-time algorithm was given for the problem but only for the case $m = 1$. In this dissertation, we propose an $O(n^2 \log n \log \log n + nm \log m)$-time algorithm for any value m, which improves the previous work even for $m = 1$. We then consider a line-constrained version of the problem in which the segments of S are all initially on the line L. Previously, an $O(n \log n)$-time algorithm was known for the case $m = 1$. We present an algorithm of $O((n+m)\log(n+m))$ time for any m, which generalizes the previous work.

Our results on this problem have been published in a conference [22]. Refer to Chapter 5 for the details.

1.3.5 Separating Overlapped Intervals on a Line

This is a general case of the spreading points problem on a line. Given n intervals on a line ℓ, we consider the problem of moving these intervals on ℓ such that after the movement no two intervals overlap and the maximum moving distance of the intervals is minimized. The difficulty for solving the problem lies in determining the order of the intervals in an optimal solution. By interesting observations, we show that it is sufficient to consider at most n "candidate" lists of ordered intervals. Further, although explicitly maintaining these lists takes $\Omega(n^2)$ time and space, by more observations and a pruning technique, we present an algorithm that can compute an optimal solution in $O(n \log n)$ time and $O(n)$ space. We also prove an $\Omega(n \log n)$ time lower bound for solving the problem, which implies the optimality of our algorithm.

Our results on this problem has been submitted to a conference and is still under review. Refer to Chapter 6 for the details.

1.4 Dissertation Outline

The rest of this dissertation is organized as follows. We present our algorithm for the balanced splitting on weighted intervals problem in Chapter 2. The algorithms for minimizing the movements of spreading points are discussed in Chapter 3. In Chapter 4, we give the results on dispersing points on intervals. Our algorithms for covering multiple barriers are described in Chapter 5. Chapter 6 presents the algorithm for the problem of separating overlapped intervals. Finally, the future work is discussed in Chapter 7.

CHAPTER 2

BALANCED SPLITTING ON WEIGHTED INTERVALS

2.1 Introduction

We consider the problem of splitting weighted intervals in a balanced way in this chapter. The results in this chapter have been published in a journal [18].

2.1.1 Problem Definitions and Our Results

Let $\mathcal{I}$ be a set of n intervals on a line L, where each interval has a non-negative weight. Given an integer $k \geq 1$, we want to find k points on L to partition L into $k+1$ segments, such that the maximum cost of these segments is minimized, where the *cost* of each segment s is the sum of the weights of the intervals in $\mathcal{I}$ that "properly" intersect s (i.e., the intersection contains more than one point). The formal definition is given below.

Let $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ be a set of n intervals on a line L, and each interval I_i has a weight $w_i \geq 0$. For simplicity, we assume L is the x-axis, and depending on the context, any real value $x \in \mathbb{R}$ is also considered as the point on L with coordinate x, and vice versa. Each interval I_i is represented as $[l_i, r_i]$ with $l_i < r_i$, where l_i is its *left endpoint* and r_i is its *right endpoint*. Note that we consider each I_i as a closed interval including both endpoints.

For an integer $k \geq 1$, consider any k points $x_1, x_2, \ldots, x_k$ on L with $x_1 < x_2 < \cdots < x_k$, and we refer to these k points as *splitters*. For simplicity of discussion, let $x_0 = -\infty$ and $x_{k+1} = +\infty$. The above k splitters partition the line L into $k+1$ *open segments*: $s_i = (x_{i-1}, x_i)$ for $i = 1, 2, \ldots, k+1$. For each segment s_i, we define its *cost* $C(s_i)$ as the sum of the weights of the intervals of $\mathcal{I}$ that intersect s_i (e.g., see Fig. 2.1). Note that since each s_i is an open segment (i.e., it does not contain its two endpoints) and each interval of $\mathcal{I}$ is closed, if an interval I_i intersects s_i, then their intersection contains

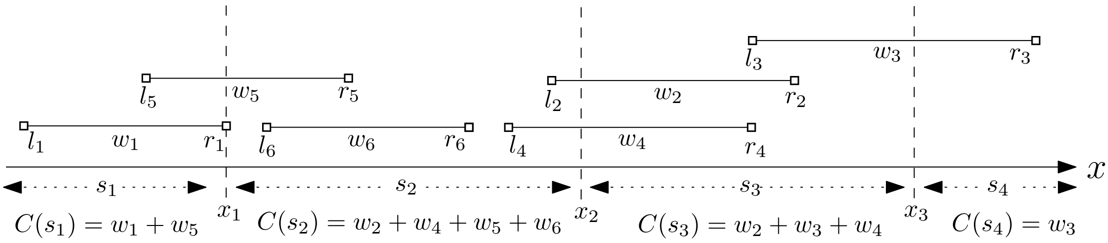


Figure 2.1. Illustrating an example of the interval splitting problem for $k = 3$: Finding three points $x_1 < x_2 < x_3$ such that the maximum value of $\{C(s_1), C(s_2), C(s_3), C(s_4)\}$ is minimized.

more than one point.

The *interval splitting* problem is to find k points/splitters $x_1, x_2, \ldots, x_k$ to partition L into $k+1$ open segments (as defined above) such that the maximum cost of all segments (i.e., $\max_{i=1}^{k+1} C(s_i)$) is minimized (e.g., see Fig. 2.1).

Previously, Le *et al.* [23] gave an $O(n \log n)$ time algorithm for a *special case* of this problem where $w_i = 1$ for each $1 \leq i \leq n$. Their algorithm, which is based on the observation that the maximum cost in any optimal solution must be an integer in $[1, n]$, does not work for our more general problem (see Section 2.4 for more discussions). In this chapter, by developing new algorithmic techniques, we solve the general case in $O(n \log n)$ time.

2.1.2 Applications and Related Work

As discussed in [23], the interval splitting problem has applications in load balancing for storing and processing data in temporal and multi-version databases. If we consider the x-axis L as the time, each interval represents a time period during which an object in databases is associated with the same value. Since an object may be associated with different values during different time periods, the task is to store and process a large number of intervals in a distributed store. To this end, one can split the intervals into "buckets" (corresponding to the segments of L in our problem) such that intervals from the same buckets can be stored in one node and processed by one core from a cluster of machines. One challenging problem is to achieve load-balancing in this process, i.e., no single node and core should store and process too many intervals. This is exactly (the special case of) our interval splitting problem. If each object has a weight, which may represent the difficulty or importance for storing and processing the object (and its corresponding intervals), then the problem becomes the general case of the interval splitting problem. Refer to [23] and the references therein for more discussions on temporal and multi-version databases.

The interval splitting problem is related to the classical interval scheduling problems. In the interval scheduling, each interval represents the time period during which a task needs to be executed. A subset of intervals is *compatible* if no two intervals overlap. One basic problem is to find a largest compatible set, and the problem can be solved by a simple greedy algorithm as shown in [7]. There are many other variations of problem; e.g., see [7, 24–27].

Since problems related to intervals are normally very fundamental, there are many powerful tools dealing with these problems, such as interval graphs [28], interval trees [29], segment trees [1], etc. Unfortunately, none of these techniques seems useful for solving our interval splitting problem.

As discussed in [23], the interval splitting problem is also related to many other problems, e.g., finding optimal splitters for a set of one dimensional points [30], the array partitioning problems [31, 32], etc.

2.1.3 Our Approach

We observe that there must exist an optimal solution in which every splitter is at the endpoint of an interval in $\mathcal{I}$. This observation implies that the objective value (i.e., the maximum cost of all segments s_i) of the optimal solution must be determined by two interval endpoints along with $-\infty$ and $+\infty$. This immediately gives $\Theta(n^2)$ *candidate* values for the optimal objective value since there are $2n$ interval endpoints. We can easily find the optimal objective value from these candidate values if we can solve the decision version of the problem: Given any value c, determine whether we can find k splitters such that the maximum cost of all segments s_i is no more than c.

Assume the $2n$ interval endpoints have already been sorted. We first present a greedy algorithm that can solve the decision version in $O(n)$ time. Then we use this algorithm to find the optimal objective value from the above candidate values. One difficulty is that since there are $\Theta(n^2)$ candidate values, computing them needs $\Omega(n^2)$ time. To reduce the running time, we manage to *implicitly* organize all the candidate values in $O(n)$ arrays and each array contains $O(n)$ elements in sorted order, and further, we give a data structure that can compute any candidate value in $O(1)$ time after $O(n)$ time preprocessing. Using this data structure and our decision algorithm, we apply a technique, called *binary search on sorted arrays* [33], to compute the optimal objective value in the above $O(n)$ sorted arrays. These efforts together lead to an $O(n \log n)$ time algorithm for solving the interval splitting problem.

The rest of this chapter is organized as follows. We introduce some notations, definitions, and observations in Section 2.2. The algorithm for the decision problem is given in Section 2.3. In Section 2.4, we solve the interval splitting problem, which is referred to as the *optimization problem*. Section 2.5 concludes this chapter.

2.2 Preliminaries

For ease of discussion, we make a general position assumption that no two intervals of $\mathcal{I}$ share the same endpoint, and our techniques can be easily adapted to the degenerate case.

We use an *open segment* to refer to a segment on L that does not include its endpoints. For any open segment s, let $\mathcal{I}(s)$ denote the set of intervals of $\mathcal{I}$ intersecting s, and let $C(s)$ denote the sum of the weights of the intervals in $\mathcal{I}(s)$ and we also call $C(s)$ the *cost* of s. For any point x on L, we let $\mathcal{I}(x)$ denote the set of intervals of $\mathcal{I}$ each of which contains x in its interior, and let $C(x)$ denote the sum of the weights of the intervals in $\mathcal{I}(x)$.

Let $X = \{x_1, x_2, \ldots, x_t\}$ be a set of points/splitters on L with $x_1 < x_2 < \cdots < x_t$, where t may or may not be equal to k. These splitters partition L into $t + 1$ open segments, and we denote by $C(X)$ the maximum cost of these open segments and $C(X)$ is referred to as the *cost* of X. We use C_{opt} to denote the cost of the set of splitters in any optimal solution of the interval splitter problem (for k splitters), and C_{opt} is also referred to as the *optimal objective value*.

Let E denote the set of all $2n$ endpoints of the intervals of $\mathcal{I}$. Due to our general position assumption, no two points of E have the same position. Let $e_1, e_2, \ldots, e_{2n}$ be the list of the points of E sorted on L from left to right.

We first prove Lemma 2.2.1. A similar observation has been made by Le *et al.* [23]

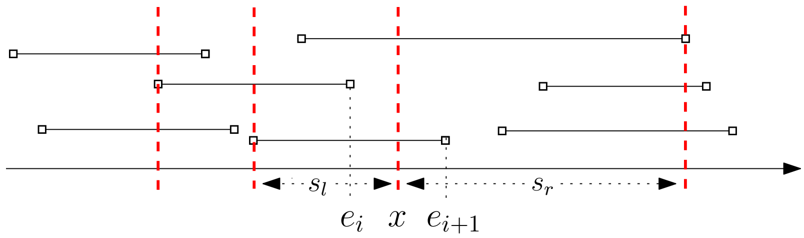


Figure 2.2. Illustrating an example for the proof of Lemma 2.2.1: There are four splitters shown with the (red) dashed vertical segments, and the splitter x is in (e_i, e_{i+1}). s_l and s_r are the two open segments bounded by x.

for the special case where the weights of all intervals of $\mathcal{I}$ are 1, and here we extend their result to the general case.

Lemma 2.2.1. *For the interval splitting problem, there must exist an optimal solution in which every splitter is at the endpoint of an interval in $\mathcal{I}$ (i.e., every splitter is in E).*

Proof. Consider any optimal solution and assume $X = \{x_1, x_2, \ldots, x_k\}$ are the set of splitters sorted on L from left to right. We assume no two splitters in X have the same position since otherwise we could consider splitters at the same position as a single splitter.

If $X \subseteq E$, then we are done with the proof. Otherwise, consider any splitter x in X but not in E (i.e., $x \in X \setminus E$). For ease of discussions, we assume $x \in (e_1, e_{2n})$. Hence, there is some i with $1 \leq i \leq k-1$ such that $x \in (e_i, e_{i+1})$. If the open interval (e_i, e_{i+1}) contains some other splitters in X, then among such splitters, we let x represent the one closest to e_i. Hence, there is no splitter in the interval (e_i, x) (e.g., see Fig. 2.2).

An easy observation is that if we move x to e_i, the value $C(X)$ does not increase. Since X is an optimal solution, we further conclude $C(X)$ does not change and we have obtained another optimal solution after x moves to e_i. Notice that in the new optimal solution, the size $|X \setminus E|$ become one less than before. If in the new optimal solution the size $|X \setminus E|$ is zero, then we are done with the proof (i.e., we have found an optimal solution in which all splitters are in E); otherwise, we repeatedly apply the above "moving technique" until $|X \setminus E|$ becomes zero. The lemma thus follows. $\square$

For any two points p and q on L, let $\overline{pq}$ be the *open* line segment whose endpoints are p and q (but $\overline{pq}$ does not include its endpoints). Recall that $\mathcal{I}(\overline{pq})$ is the set of intervals of $\mathcal{I}$ that intersect $\overline{pq}$, and $C(\overline{pq})$ is the sum of the weights of the intervals in $\mathcal{I}(\overline{pq})$.

From now on, we let E also include the two infinite points $-\infty$ and $+\infty$ on L. Let S_E consists of all values $C(\overline{pq})$ for any two points p and q in E. Lemma 2.2.1 implies the following corollary.

Corollary 2.2.2. $C_{opt} \in S_E$.

Proof. By Lemma 2.2.1, there is an optimal solution in which the set X of splitters is a subset of E. Hence, $C_{opt} = C(X)$. The splitters of X partition L into open segments and there must be a segment s_i such that $C(X) = C(s_i)$. Clearly, both endpoints of s_i are in E. By the definition of S_E, $C(s_i) = C_{opt}$ must be in S_E. □

For any value c, if there exists a set X of at most k splitters such that $C(X) \leq c$, then we call c a *feasible value* and call X a *feasible splitter set* with respect to c. For any given value c, the *decision version* of our interval splitting problem is to determine whether c is a feasible value, and if yes, find a feasible splitter set. For differentiation, we refer to our original interval splitting problem as the *optimization version*.

In the sequel, we will first present our algorithm for the decision problem in Section 2.3 and then solve the optimization problem in Section 2.4.

2.3 The Decision Problem

In this section, we solve the decision version of the problem. Our algorithm runs in $O(n)$ time after the points in E are sorted. Note that Le *et al.* [23] also gave a linear time algorithm (after the points in E are sorted), but their algorithm only works for the special case. Our algorithm solves the general case. In the following, we assume the points of E have been sorted.

Our algorithm uses the greedy approach. Let c be any given value. If c is a feasible value, the algorithm will find from left to right at most k splitters $x_1, x_2, \ldots$, that are feasible for c; otherwise it will report that c is not feasible.

Recall that for any point $x \in L$, $\mathcal{I}(x)$ is the set of intervals of $\mathcal{I}$ each of which contains x in its interior, and $C(x)$ is the sum of the weights of the intervals in $\mathcal{I}(x)$.

We first give the following lemma, which will be useful later for proving the correctness of our algorithm.

Lemma 2.3.1. *If there is a point q on L with $C(q) > c$, then c is not a feasible value.*

Proof. Assume to the contrary that c is a feasible value. Let $X = \{x_1, x_2, \ldots, x_k\}$ be a feasible splitter set. Thus we have $C(X) \leq c$. Let $x_0 = -\infty$ and $x_{k+1} = +\infty$. Assume q is in $[x_{i-1}, x_i)$ for some index i. Let s_i be the open interval (x_{i-1}, x_i). Depending on whether $q = x_{i-1}$, there are two cases.

1. If $q \neq x_{i-1}$, then $q \in s_i$. Based on their definitions, we have $\mathcal{I}(q) \subseteq \mathcal{I}(s_i)$, and thus $C(q) \leq C(s_i)$. Note that $C(X) \geq C(s_i)$. Since $C(q) > c$, we obtain $C(X) \geq C(s_i) \geq C(q) > c$, which contradicts with that $C(X) \leq c$.

2. If $q = x_{i-1}$, then q is not in s_i. Let q' be a point to the right of q and infinitesimally close to q. Clearly, $q' \in s_i$. Further, it always holds that $C(q') \geq C(q)$. Consequently, we obtain $C(X) \geq C(s_i) \geq C(q') \geq C(q) > c$, which again contradicts with $C(X) \leq c$.

The lemma is thus proved. □

We first describe the main idea of our algorithm and then flesh out the details. The algorithm starts with setting x_0 to $-\infty$ (note that x_0 is not a splitter). Assume x_{i-1} has already been computed for any $1 \leq i \leq k$. Our algorithm sweeps a point x from x_{i-1} to the right as far as possible to find x_i. For any $x > x_{i-1}$, recall that $C(\overline{x_{i-1}x})$ is the sum of the weights of the intervals in $\mathcal{I}$ that intersect the open segment $\overline{x_{i-1}x} = (x_{i-1}, x)$. During the rightward sweeping of x, as long as $C(\overline{x_{i-1}x}) \leq c$, we continue to move x rightwards. But if moving x rightwards will make the value $C(\overline{x_{i-1}x})$ larger than c, then we stop and put the next splitter x_i at the current position of x; if the above situation happens when $x = x_{i-1}$, then we terminate the algorithm and conclude that c is not a feasible solution. If x has moved to the right of all intervals of $\mathcal{I}$, then we terminate the algorithm and conclude that c is feasible value. In addition, if the algorithm has already put k splitters (i.e., $k = i - 1$) but still need to put the next splitter x_{k+1}, then we conclude that c is not a feasible solution and terminate the algorithm. The details on how to implement the algorithm are given below.

Our algorithm will maintain an *invariant* that each splitter (i.e., x_i for $1 \leq i \leq k$) computed by the algorithm is at the left endpoint of an interval of $\mathcal{I}$. Assume x_{i-1} has just been computed and x is at x_{i-1}. In order to compute the value $C(\overline{x_{i-1}x})$ during the rightward sweeping of x, we need to know the value $C(x_{i-1})$. We assume $C(x_{i-1})$ is already known when x is at x_{i-1}. Initially when $i = 1$, we set $x_{i-1} = -\infty$ and $C(-\infty) = 0$. Further, during the sweeping of x, we will maintain the value $C(x)$, which will be used to compute $C(x_i)$ once the next splitter x_i is determined. We will show that after x_i is determined, x_i is at the left endpoint of an interval and $C(x_i)$ is computed correctly.

During the sweeping of x, an *event* happens when x encounters a point of E. Suppose we have just computed x_{i-1}. For the case where $i \geq 2$, before we sweep x rightwards, we first process this *beginning event* for $x = x_{i-1}$ as follows.

For any interval $I \in \mathcal{I}$, we use $w(I)$ to denote its weight.

Since $i \geq 2$, by our algorithm invariant, x_{i-1} is the left endpoint of an interval, denoted by I. Also recall that $C(x_{i-1})$ is known. If $C(x_{i-1}) + w(I) > c$, then we conclude that c is not a feasible solution and terminate the algorithm. The correctness is proved in the following lemma.

Lemma 2.3.2. *If $C(x_{i-1}) + w(I) > c$, then c is not a feasible value.*

Proof. Consider any point q to the right of x_{i-1} and infinitesimally close to x_{i-1}. Since x_{i-1} is the left endpoint of I, it holds that $\mathcal{I}(q) = \mathcal{I}(x_{i+1}) \cup \{I\}$, and thus, $C(q) = C(x_{i-1}) + w(I)$. It follows that $C(q) > c$. By Lemma 2.3.1, c is not a feasible value. □

If $C(x_{i-1}) + w(I) \leq c$, then we are "safe" to move x rightwards. We also set $C(\overline{x_{i-1}x}) = C(x) = C(x_{i-1}) + w(I)$. One can verify that the above values are correct when x is moving rightwards before the next event happens. This finishes our processing on the beginning event for $x = x_{i-1}$.

Below, we discuss the general events after the beginning event. Suppose the next event is at a point e in E (with $x_{i-1} < e < +\infty$) and assume that $C(x)$ and $C(\overline{x_{i-1}x})$ have been correctly maintained for x right before x arrives at e. We process the event e as follows. Let I be the interval for which e is its endpoint. Depending on whether e is the right or left endpoint of I, there are two cases.

1. If e is the right endpoint of I, then we update $C(x)$ by setting $C(x) = C(x) - w(I)$ and continue to move x rightwards and proceed on the next event after e. Note that we do not need to change the value $C(\overline{x_{i-1}x})$.

   If e is the rightmost point of E, then we terminate the algorithm and conclude that c is a feasible value, and the set of $i-1$ splitters $x_1, x_2, \ldots, x_{i-1}$ that have been computed so far is a feasible splitter set.

2. If e is the left endpoint of I, then we first check whether $C(\overline{x_{i-1}x}) + w(I) \leq c$. If yes, we set $C(x) = C(x) + w(I)$ and $C(\overline{x_{i-1}x}) = C(\overline{x_{i-1}x}) + w(I)$, and continue to move x rightward and proceed on the next event after e.

   If $C(\overline{x_{i-1}x}) + w(I) > c$, we need to put the next splitter x_i at e. But if $i = k+1$, then we terminate the algorithm and conclude that c is not a feasible value because we are only allowed to have k splitters. If $i < k+1$, then we let $x_i = e$ and proceed on finding the next splitter x_{i+1}. Note that x_i is at the left endpoint of I, which maintains the algorithm invariant. Also, it is easy to see that $C(x_i) = C(x)$.

This finishes the description of our algorithm. For the running time, since the points of E have already been sorted, after processing each event, we can find the next event point in constant time. Also, processing each event takes only constant time. Hence, the total time of the algorithm is $O(n)$. The correctness of the algorithm can be seen from Lemma 2.3.2 as well as the fact that our algorithm always tries to push the splitters rightward on L as far as possible.

As a summary, we have the following result.

Theorem 2.3.3. *Suppose the endpoints of all intervals in $\mathcal{I}$ have been sorted. The decision version of the interval splitting problem can be solved in $O(n)$ time.*

2.4 The Optimization Problem

In this section, we solve the optimization version of the interval splitting problem, with the help of Corollary 2.2.2 and Theorem 2.3.3. In the following, we refer to our algorithm for the decision problem in Theorem 2.3.3 as the *decision algorithm.*

Recall that C_{opt} is the optimal objective value. If we know the value C_{opt}, then we can compute an optimal solution by using our decision algorithm. Specifically, we apply our decision algorithm on $c = C_{opt}$, and the algorithm will find a feasible splitter set, which is an optimal solution. Hence, to solve the optimization problem, the key is to compute C_{opt}, which is our focus below.

Note that in the special case where the weight of each interval of $\mathcal{I}$ is 1, an easy observation is that C_{opt} must be an integer in $[1, n]$. Thus, using the decision algorithm, we can easily compute C_{opt} in $O(n \log n)$ time by doing binary search on the integer sequence from 1 to n. This is exactly the approach used in [23] (by using their own decision algorithm, which works only for the special case). In our general problem, however, this approach does not work because C_{opt} may not be an integer. We propose a new approach, as follows.

2.4.1 Computing the Optimal Objective Value C_{opt}

Recall that the set S_E consists of all values $C(\overline{pq})$ for any two points p and q in E. By Corollary 2.2.2, we have $C_{opt} \in S_E$. One straightforward way to compute C_{opt} is to first compute all values in the set S_E and sort them. Then, using our decision algorithm in Theorem 2.3.3, we can compute C_{opt} by doing binary search on the sorted list of the values in S_E. However, since $|S_E| = \Theta(n^2)$, this approach takes $\Omega(n^2)$ time. In the following, we give an $O(n \log n)$ time algorithm.

Recall that E also includes $-\infty$ and $+\infty$. We first organize the values in S_E into $O(n)$ sorted arrays and each array has $O(n)$ elements. Note that our algorithm does not do this organization explicitly.

Let $e_0, e_1, \ldots, e_{2n+1}$ be the list of the values of E sorted on L from left to right, with $e_0 = -\infty$ and $e_{2n+1} = +\infty$. For any i and j with $0 \leq i < j \leq 2n + 1$, define $w(i, j) = C(\overline{e_i e_j})$. Clearly, $S_E = \{w(i, j) \mid 0 \leq i < j \leq 2n + 1\}$. Below is a self-evident observation that shows a monotonicity property of $w(i, j)$.

Observation 2.4.1. *For any i, if $i < j_1 \leq j_2$, then $w(i, j_1) \leq w(i, j_2)$.*

For each $i = 0, 1, \ldots, 2n + 1$, we define an array $A_i[0 \cdots 2n + 1]$ of $2n + 2$ elements as follows. For each j with $0 \leq j \leq 2n + 1$, define $A_i[j]$ to be $w(i, j)$ if $i < j$ and 0 otherwise. By Observation 2.4.1, elements in each array A_i are sorted in ascending order. It is not difficult to see that S_E is the union of all elements in the arrays A_i, $0 \leq i \leq 2n+1$, i.e., $S_E = \bigcup_{i=0}^{2n+1} A_i$.

Since $C_{opt} \in S_E$, our goal is to find C_{opt} in $\bigcup_{i=0}^{2n+1} A_i$. To this end, although we cannot afford to explicitly compute all elements of these arrays, based on the following Lemma 2.4.2, with linear time preprocessing, we can obtain any element of these arrays in constant time whenever we need it.

Lemma 2.4.2. *With $O(n)$ time preprocessing, for any query (i, j) with $i < j$, we can compute the value $w(i, j) = A_i[j]$ in constant time.*

Before proving Lemma 2.4.2, we show how to compute C_{opt} with the help of Lemma 2.4.2. We use a technique, called *binary search on sorted arrays*, which was developed in [33]. We first briefly discuss this technique.

Assume there is a "black-box" decision procedure σ available such that given any value α, σ can report whether α is a feasible value in $O(T)$ time, and further, if α is a feasible value, then any value larger than α is also a feasible value. Given a set of M arrays B_i, $1 \leq i \leq M$, each containing N elements in sorted order, the goal is to find the smallest feasible value δ in $\bigcup_{i=1}^{M} B_i$. Suppose given its indices, any element of these arrays can be obtained in constant time. An algorithm is presented in [33] with the following result.

Lemma 2.4.3. [33] *The smallest feasible value δ in $\bigcup_{i=1}^{M} B_i$ can be found in $O((M + T)\log(MN))$ time.*

For solving our problem, we can use the above result to find C_{opt} in $\bigcup_{i=0}^{2n+1} A_i$ as follows. The following observation is self-evident.

Observation 2.4.4. *If a value c is a feasible value for the decision problem, then any value larger than c is also a feasible value.*

Hence, C_{opt} is the smallest feasible value in $\bigcup_{i=0}^{2n+1} A_i$. Our linear time decision algorithm in Theorem 2.3.3 can play the role of the black-box σ with $T = O(n)$. Further, we have already shown that given any i and j, we can compute the element $A_i[j]$ in constant time. Therefore, we can apply the technique in Lemma 2.4.3 (with $M = N = 2n+2$ and $T = O(n)$) to compute C_{opt} in $O(n \log n)$ time.

In summary, we have the following result.

Theorem 2.4.5. *The optimization version of the interval splitting problem can be solved in $O(n \log n)$ time.*

2.4.2 Proving Lemma 2.4.2

It remains to prove Lemma 2.4.2. Consider any query (i, j) with $i < j$. Our goal is to compute $w(i, j) = C(\overline{e_i e_j})$. We begin with some observations.

Recall that $e_0, e_1, \ldots, e_{2n+1}$ are the sorted list of points of E. For each t with $0 \leq t \leq 2n + 1$, define $\mathcal{I}_t$ to be the set of intervals of $\mathcal{I}$ whose left endpoints are strictly to the left of e_t. Recall that for any point x on L, $\mathcal{I}(x)$ is the set of intervals of $\mathcal{I}$ each of which contains x in its interior. Also recall that $\mathcal{I}(\overline{e_i e_j})$ is the set of intervals of $\mathcal{I}$ that intersect the open segment $\overline{e_i e_j}$. We have the following lemma.

Lemma 2.4.6. $\mathcal{I}(\overline{e_i e_j}) = \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$ *and* $\mathcal{I}(e_i) \cap (\mathcal{I}_j \setminus \mathcal{I}_i) = \emptyset$.

Proof. We first prove $\mathcal{I}(\overline{e_i e_j}) = \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$. To this end, we show below that any interval in $\mathcal{I}(\overline{e_i e_j})$ must be in $\mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$, and vice versa.

1. Consider any interval $I \in \mathcal{I}(\overline{e_i e_j})$. We prove that I must be in $\mathcal{I}(e_i) \cup \mathcal{I}_j \setminus \mathcal{I}_i$.

   Let l and r be the left and right endpoints of I, respectively. By definition, I intersects the open segment $\overline{e_i e_j}$. Hence, $l < e_j$, implying that $I \in \mathcal{I}_j$. If $I \notin \mathcal{I}_i$, it is vacuously true that $I \in \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$. Otherwise, it must be that $l < e_i$. Since I intersects the open segment $\overline{e_i e_j}$, we can also get $r > e_i$. Therefore, it holds that $l < e_i < r$, implying that e_i is contained in the interior of I, and thus $I \in \mathcal{I}(e_i)$.

   Therefore, in any case, we obtain $I \in \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$.

2. Consider any interval $I \in \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$. We prove that I must be in $\mathcal{I}(\overline{e_i e_j})$. Let l and r be the left and right endpoints of I, respectively.

   If $I \in \mathcal{I}_j \setminus \mathcal{I}_i$, then due to $I \in \mathcal{I}_j$, we obtain $l < e_j$, and due to $I \notin \mathcal{I}_i$, we obtain $e_i \leq l$. Hence, we have $e_i \leq l < e_j$. Since $l < r$, I must intersect the open segment $\overline{e_i e_j}$, and thus $I \in \mathcal{I}(\overline{e_i e_j})$.

   If $I \notin \mathcal{I}_j \setminus \mathcal{I}_i$, then I must be in $\mathcal{I}(e_i)$, implying that $e_i \in (l, r)$. Therefore, I must intersect the open segment $\overline{e_i e_j}$, and $I \in \mathcal{I}(\overline{e_i e_j})$.

The above proves that $\mathcal{I}(\overline{e_ie_j}) = \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$.

Next, we show that $\mathcal{I}(e_i) \cap (\mathcal{I}_j \setminus \mathcal{I}_i) = \emptyset$. Indeed, for any interval $I = [l, r] \in \mathcal{I}_j \setminus \mathcal{I}_i$, as discussed above, it holds that $e_i \leq l < e_j$, implying that e_i cannot be in the interior of I, and thus, $I \notin \mathcal{I}(e_i)$. On the other hand, for any interval $I = [l, r] \in \mathcal{I}(e_i)$, since e_i is in the interior of I, we have $l < e_i$; thus, I must be in $\mathcal{I}_i$, implying that I cannot be in $\mathcal{I}_j \setminus \mathcal{I}_i$.

The lemma thus follows. □

The preceding lemma implies the following approach for computing the value $C(\overline{e_ie_j})$. For each t with $0 \leq t \leq 2n+1$, let C_t be the sum of the weights of the intervals in $\mathcal{I}_t$. By Lemma 2.4.6, we can obtain $C(\overline{e_ie_j}) = C(e_i) + (C_j - C_i)$. Hence, if the values $C(e_i)$, C_j, and C_i are already known, we can compute $C(\overline{e_ie_j})$ in constant time. In the sequel, we present an algorithm that can compute $C(e_t)$ and C_t for all $t = 0, 1, \ldots, 2n+1$ in $O(n)$ time. The algorithm is similar to our decision algorithm in Section 2.3 (the decision algorithm can compute $C(e_t)$, but here we also need to compute C_t).

The algorithm sweeps a point x from $-\infty$ to $+\infty$. An event happens when x encounters a point, say, e_t, in E, and for processing the event, we will compute $C(e_t)$ and C_t. During the sweeping of x, we will maintain two values for x: $C(x)$, i.e., the sum of the weights of the intervals of $\mathcal{I}$ that contain x in their interior, and $C'(x)$, which is the sum of the weights of the intervals whose left endpoints are strictly to the left of x.

Initially, when $x = -\infty$, we have $C(x) = C'(x) = 0$. Consider a general step that the next event is at e_t. We assume that the values $C(x)$ and $C'(x)$ have been correctly maintained right before x arrives at e_t. Note that e_t is an endpoint of an interval of $\mathcal{I}$, and let I denote the interval (and let $w(I)$ be the weight of I). Depending on whether e_t is the left or the right endpoint of e_t, there are two cases.

If e_t is the right endpoint of I, we first set $C(e_t) = C(x) - w(I)$ and $C_t = C'(x)$. Then we update $C(x) = C(x) - w(I)$, and we do not need to change $C'(x)$. One can verify that all these values have been correctly computed. We then proceed on the next event after e_t.

If e_t is the left endpoint of I, then we set $C(e_t) = C(x)$ and $C_t = C'(x)$. We also update $C(x) = C(x) + w(I)$ and $C'(x) = C'(x) + w(I)$ because once x crosses e_t, e_t is strictly to the left of x. We proceed on the next event after e_t.

The algorithm is done once x passes the rightmost point of E. Since the points of E have already been sorted, the algorithm runs in $O(n)$ time.

As a summary, in $O(n)$ time we can compute $C(e_t)$ and C_t for all $t = 0, 1, \ldots, 2n+1$, after which, given any query (i, j) with $i < j$, we can compute $w(i, j)$ in constant time. Lemma 2.4.2 is thus proved.

2.5 Conclusions

In this chapter, we present an efficient algorithm for solving the interval splitting problem. While the previous work [23] only deals with the special/unweighted case, our algorithm works for the general/weighted case. Besides its applications in load balancing for storing and processing data in temporal and multi-version databases, the interval splitting problem itself is an interesting and basic problem on intervals. Our techniques may be used for solving other related problems as well.

CHAPTER 3

MINIMIZING THE MOVEMENTS OF SPREADING POINTS

3.1 Introduction

We consider the following *points-spreading* problem in this chapter. The results in this chapter have been published in a conference [19].

3.1.1 Problem Definitions and Our Results

Given a set P of n points sorted on a line L and a distance value $\delta \geq 0$, we wish to move the points of P along L such that the distance of any two points of P is at least δ and the maximum movement of all points of P is minimized. The above is the *line version*. We also consider the *cycle version* of the problem, where all points of P are given sorted cyclically on a cycle (one may view C as a simple closed curve). We wish to move the points of P on C such that the distance of any two points of P along C is at least δ and the maximum movement of all points of P along C is minimized. Note that since C is a cycle, the distance of any two points of C is defined to be the length of the shortest path on C between the two points.

Both versions of the problem have been studied before. By modeling them as linear programming problems (with n variables and $\Theta(n)$ constraints), Dumitrescu and Jiang [34] gave the first-known polynomial-time algorithms for both problems. Since there only exist weakly polynomial-time algorithms for linear programming [35, 36], it would be interesting to design strongly polynomial-time algorithms for the points-spreading problem. In this chapter, we solve both versions of the problem not only in strongly polynomial time but also in $O(n)$ time (which is optimal). Our algorithms are based on a greedy strategy.

In addition, we consider a somewhat related problem, called the *facility-location movement* problem, defined as follows. Suppose we have a set of k "server" points and another set of n "client" points sorted on L. We wish to move all servers and all clients on L such that each client co-locates with a server and the maximum moving distance of all servers and clients is minimized. Dumitrescu and Jiang [34] solved this problem in $O((n+k)\log(n+k))$ time. We present an $O(n+k)$ time algorithm based on their approach.

3.1.2 Related Work

The 2D version of the points-spreading problem was proposed by Demaine et al. [37] (also called "movement to independence" problem in [34, 37]). The problem in 2D is NP-hard and an approximation algorithm was given in [37]; the algorithm was improved later by Dumitrescu and Jiang [34].

The points-spreading problem is related to the points dispersion problems which involve arranging a set of points as far away from each other as possible subject to certain constraints. For example, Fiala et al. in [38] studied such a problem in which one wants to place n given points, each inside its own, prespecified disk, with the objective of maximizing the distance between the closest pair of these points. The problem was shown to be NP-hard [38]. Approximation algorithms were given for this problem by Cabello [39]. Dumitrescu and Jiang [40] gave improvement on the approximation algorithms and also proposed algorithms for the problem in high-dimensional spaces. In fact, Fiala et al. [38] studied the dispersion problems on a more general problem settings. Another variation of the dispersion problems is to select a subset of facilities from a set of given facilities to maximize the minimum distance (or some other distance function) among all pairs of selected facilities [41, 42]. The problem is generally NP-hard (e.g., in 2D) but polynomial time algorithms are available in the one-dimensional space [41, 42]. In addition, Chandra and Halldórsson [43] studied dispersion problems in other problem settings.

The facility-location movement problem was first introduced by Demaine et al. [37] in graphs, which was proved to be NP-hard. A 2-approximation algorithm was presented in [37] for this problem in graphs, and later it was shown that the 2-approximation ratio cannot be improved unless P=NP [44]. Dumitrescu and Jiang [34] studied the geometric version of this problem in the plane, and they showed that the problem is NP-hard to approximate within 1.8279. Fixed parameter algorithms (with k as the parameter) were also given in [34].

3.1.3 Our Approach

For solving the line version of the points-spreading problem, essentially we first solve a "one-direction" case of the problem in which points are only allowed to move rightwards, by using a simple greedy algorithm. Suppose d is the maximum movement in the solution of the above one-direction case. Then, we show that an optimal solution to the original problem can be obtained by shifting each point of P leftwards by the distance $d/2$.

For solving the cycle version of the problem, essentially we also first solve a one-direction case in which points are only allowed to move counterclockwise on C. If d is the maximum movement in the solution of the one-direction case, then we also show that an optimal solution to the original problem can be obtained by shifting each point of P clockwise by $d/2$. However, unlike the line version, the one-direction case of the problem becomes more difficult on the cycle. One straightforward idea is to cut the cycle C at a point of P (and extend C as a line) and then apply the algorithm for the one-direction case of the line version. However, the issue is that the last point may be too close to or even "cross" the first point if we put all points back on C. By observations, we show that if such a case happens, we can run the line-version algorithm for another round and the second round is guaranteed to find an optimal solution. Overall, the algorithm is still simple, but it is challenging to discover the idea and prove the correctness.

For solving the facility-location movement problem, Dumitrescu and Jiang [34] presented an $O((n+k)\log(n+k))$ time algorithm using dynamic programming. By discovering a monotonicity property on the dynamic programming, we improve Dumitrescu and Jiang's algorithm to $O(n+k)$ time.

The rest of this chapter is organized as follows. In Section 3.2, we present our algorithm for the line version of the points-spreading problem. The cycle version of the problem is solved in Section 3.3. Section 3.4 discusses our solution for the facility-location movement problem.

3.2 The Line Version of the Points-Spreading Problem

In the line version, the points of P are given sorted on the line L. Without loss of generality, we assume L is the x-axis and $P = \{p_1, p_2, \ldots, p_n\}$ are sorted by their x-coordinates from left to right. For each $i \in [1, n]$, let x_i denote the location (or x-coordinate) of p_i on L. For any two locations x and x' of L, denote by $|xx'|$ the distance between x and x', i.e., $|xx'| = |x - x'|$.

Our goal is to move each point $p_i \in P$ to a new location x'_i on L such that the distance of any pair of two points of P is at least δ and the maximum moving distance, i.e., $\max_{1 \le i \le n} |x_i x'_i|$, is minimized. For simplicity of discussion, we make a general position assumption that no two points of P are at the same location in the input. The degenerate case can also be handled by our techniques but the discussions would be more tedious.

We refer to a *configuration* as a specification of the location of each point p_i of P on L. For example, in the input configuration each p_i is at x_i. Let F_0 denote the input configuration. A configuration is *feasible* if the distance between any pair of points of P is at least δ.

Denote by d_{opt} the maximum moving distance in any optimal solution. If the input configuration F_0 is feasible, then we do not need to move any point, implying that $d_{opt} = 0$. Since the points of P are sorted, we can check whether F_0 is feasible in $O(n)$ time by checking the distance between every adjacent pair of points of P. If F_0 is not feasible, then $d_{opt} > 0$. In the following, we assume F_0 is not feasible, and thus $d_{opt} > 0$.

We first present some observations, based on which our algorithm will be developed.

3.2.1 Observations

For any two indices $i < j$ in $[1, n]$, define

$$w(i, j) = (j - i) \cdot \delta - |x_i x_j|.$$

As discussed in Dumitrescu and Jiang [34], there exists an optimal solution in which the order of all points of P is the same as that in the input configuration F_0. Based on this property, we prove Lemma 3.2.1 regarding the value d_{opt}.

Lemma 3.2.1. $d_{opt} \geq \max_{1 \leq i < j \leq n} \frac{w(i,j)}{2}$.

Proof. Consider any optimal solution OPT in which the order of all points of P is the same as that in F_0. For each $1 \leq i \leq n$, let x_i^* be the location of p_i in OPT.

Consider any i and j with $1 \leq i < j \leq n$. Our goal is to prove $d_{opt} \geq w(i,j)/2$.

Since the points of P in OPT have the same order as in F_0, for each k with $i < k \leq j$, we have $|x_{k-1}^* x_k^*| \geq \delta$ because OPT is a feasible solution. Hence, $|x_i^* x_j^*| = \sum_{k=i+1}^{j} |x_{k-1}^* x_k^*| \geq (j-i) \cdot \delta$.

If $|x_i^* x_j^*| - |x_i x_j| \leq 0$, then $|x_i x_j| \geq |x_i^* x_j^*| \geq (j-i) \cdot \delta$. Thus, $w(i,j) \leq 0$. Since $d_{opt} > 0$, $d_{opt} \geq w(i,j)/2$ holds.

If $|x_i^* x_j^*| - |x_i x_j| > 0$, then the difference of $|x_i^* x_j^*|$ and $|x_i x_j|$ are due to the moving of p_i and p_j. It is not difficult to see that $\max\{|x_i x_i^*|, |x_j x_j^*|\} \geq (|x_i^* x_j^*| - |x_i x_j|)/2$ (the equality happens when p_i moves leftwards by distance $(|x_i^* x_j^*| - |x_i x_j|)/2$ and p_j moves rightwards by the same distance). Since $d_{opt} \geq \max\{|x_i x_i^*|, |x_j x_j^*|\}$, it holds that $d_{opt} \geq (|x_i^* x_j^*| - |x_i x_j|)/2$. Due to $|x_i^* x_j^*| \geq (j-i) \cdot \delta$, we obtain that $d_{opt} \geq w(i,j)/2$.

The lemma thus follows. □

Lemma 3.2.2. *If there exist i and j with $1 \leq i < j \leq n$ and a feasible configuration F' in which each point $p_k \in P$ moves rightwards to x'_k (i.e., $x_k \leq x'_k$) such that $w(i,j) = \max_{1 \leq k \leq n} |x_k x'_k|$, then we can obtain an optimal solution by shifting each point of P in F' leftwards by distance $w(i,j)/2$.*

Proof. Let F'' denote the configuration obtained by shifting each point of P in F' leftwards by distance $w(i,j)/2$.

Consider any point $p_k \in P$. Let x''_k denote the location of p_k in F'', i.e., $x''_k = x'_k - w(i,j)/2$. In order to prove that F'' is an optimal solution, by Lemma 3.2.1, it is sufficient to show that $|x_k x''_k| \leq w(i,j)/2$, as follows.

Indeed, since $0 \leq x'_k - x_k \leq w(i,j)$, i.e., x'_k is to the right of x_k at most $w(i,j)$, after p_k is moved leftwards by $w(i,j)/2$ to x''_k, x''_k must be within distance $w(i,j)/2$ from x_k. Hence, $|x_k x''_k| \leq w(i,j)/2$. The lemma thus follows. □

We call a feasible configuration that satisfies the condition in Lemma 3.2.2 a *canonical configuration* (such as F' in Lemma 3.2.2). Due to Lemma 3.2.2, to solve the problem

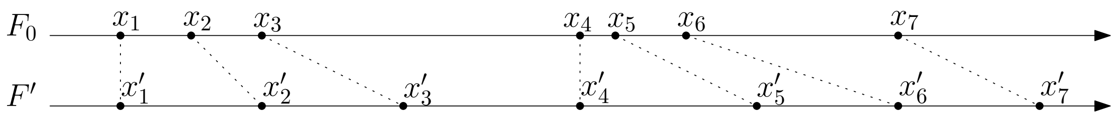


Figure 3.1. Illustrating our algorithm for computing the configuration F.

in linear time, it is sufficient to find a canonical configuration in linear time, which is our focus below.

3.2.2 Computing a Canonical Configuration

In this section, we present a linear-time algorithm that can find a canonical configuration. Comparing with the original problem, now we only need to consider the rightward movements.

Initially, we set $x'_1 = x_1$. Then we consider the rest of the points $p_2, p_3, \ldots, p_n$ from left to right. For each i with $2 \leq i \leq n$, suppose we have already moved p_{i-1} to x'_{i-1}. Then, we set $x'_i = \max\{x_i, x'_{i-1} + \delta\}$, and move p_i to x'_i. Refer to Fig. 3.1 for an example. The algorithm finishes after all points of P have been considered. Clearly, the algorithm runs in $O(n)$ time. Let F' denote the resulting configuration (i.e., each p_i is at x'_i).

In the following lemma, we show that F' is a canonical configuration.

Lemma 3.2.3. *F' is a canonical configuration.*

Proof. First of all, based on our way of setting x'_i for $i = 1, 2, \ldots, n$, it can be easily seen that every two points of P in F' are at least δ away from each other. Thus, F' is a feasible configuration. Note that $x'_i \geq x_i$ for any $i \in [1, n]$.

Next, we show that there exist i and j with $1 \leq i < j \leq n$ such that $w(i, j) = d_{max}$, where $d_{max} = \max_{1 \leq k \leq n} |x_k x'_k|$.

Recall that $d_{max} > 0$. Suppose the moving distance of p_j is the maximum, i.e., $d_{max} = |x_j x'_j|$. Let i be the largest index such that $i < j$ and p_i does not move in the algorithm (i.e., $x_i = x'_i$). Note that such a point p_i must exist as $x_1 = x'_1$ and $x'_j > x_j$.

For any point $p_k \in P$, if p_k is moved (rightwards) in F' (i.e., $x_k < x'_k$), then according to our way of setting x'_k, it must hold that $x'_k - x'_{k-1} = \delta$. By the definition of i, for each point p_k with $k \in [i+1, j]$, p_k is moved in F', and thus $x'_k - x'_{k-1} = \delta$. Therefore, we obtain

$$|x'_i x'_j| = x'_j - x'_i = \sum_{i+1 \leq k \leq j} (x'_k - x'_{k-1}) = (j-i) \cdot \delta.$$

Since $x'_i = x_i$ and $x_j < x'_j$, we have $|x_i x'_j| = |x_i x_j| + |x_j x'_j|$. Hence, $d_{max} = |x_j x'_j| = |x_i x'_j| - |x_i x_j| = (j-i) \cdot \delta - |x_i x_j| = w(i,j)$.

This proves the lemma. □

Combining Lemmas 3.2.2 and 3.2.3, we conclude this section with the following theorem.

Theorem 3.2.4. *The line version of the points-spreading problem is solvable in $O(n)$ time.*

Remark: One may verify that our algorithm for computing the canonical configuration F' essentially solves the following *one-direction case* of the line version problem: Move the points of P rightwards such that any pair of points of P are at least δ away from each other and the maximum moving distance of all points of P is minimized.

3.3 The Cycle Version of the Points-Spreading Problem

In the cycle version, the points of $P = \{p_1, p_2, \ldots, p_n\}$ are on a cycle C sorted cyclically, say, in the counterclockwise order. We use $|C|$ to denote the length of C. For any two locations x and x' on C, the distance between x and x', denoted by $|xx'|$, is the length of the shortest path between x and x' on C. Clearly, $|xx'| \leq |C|/2$. For each $i \in [1, n]$, we use x_i to denote the location of p_i on C in the input. Our goal is to move each point $p_i \in P$ to a new location x'_i such that the distance of any pair of two points of P on C is at least δ and the maximum moving distance, i.e., $\max_{1 \leq i \leq n} |x_i x'_i|$, is minimized.

We assume $|C| \geq \delta \cdot n$ since otherwise there would be no solution. Again, for simplicity of discussion, we make a general position assumption that no two points of P are at the same location on C in the input.

As in the line version, we refer to a *configuration* as a specification of the location of each point of P on C. A configuration is *feasible* if the distance between any pair of points of P is at least δ. Let F_0 denote the input configuration.

Denote by d_{opt} the maximum moving distance in any optimal solution. If F_0 is feasible, then $d_{opt} = 0$. We can also check whether F_0 is feasible in $O(n)$ time. If F_0 is not feasible, then $d_{opt} > 0$. In the following, we assume F_0 is not feasible, and thus $d_{opt} > 0$.

To solve the cycle version of the problem, we extend our algorithm (and observations) for the line version in Section 3.2. Namely, we first move all points of P on C counterclockwise to obtain a "canonical configuration", and then shift all points clockwise. However, as will be seen later, the problem becomes much more difficult on the cycle.

Consider any two locations x and x' on C. We define $C(x, x')$ as the portion of C from x to x' counterclockwise. We use $|C(x, x')|$ to denote the length of $C(x, x')$. Note that $|xx'| = \min\{|C(x, x')|, |C(x', x)|\}$.

As in the line version, we first give some observations, based on which our algorithms will be developed.

3.3.1 Observations

For any two indices $i \neq j$ in $[1, n]$, define

$$w(i,j) = \big[(n + j - i) \mod n\big] \cdot \delta - |C(x_i, x_j)|.$$

In words, if $i < j$, then $w(i,j) = (j - i) \cdot \delta - |C(x_i, x_j)|$; otherwise, $w(i,j) = (n + j - i) \cdot \delta - |C(x_i, x_j)|$. Since $|C| \geq \delta \cdot n$, it can be verified that $w(i,j) \leq |C|$.

As discussed in [34], there exists an optimal solution in which the order of all points of P is the same as that in the input configuration F_0. Using this property, we prove Lemma 3.3.1, which is analogous to Lemma 3.2.2 for the line version.

Lemma 3.3.1. $d_{opt} \geq \max_{1 \leq i,j \leq n} \frac{w(i,j)}{2}$.

Proof. Consider any optimal solution OPT in which the order of all points of P is the same as that in the input configuration F_0. For each $1 \leq k \leq n$, let x_k^* be the location of x_k in OPT.

Consider any two indices $i \neq j$ in $[1, n]$. To prove the lemma, the goal is to show that $d_{opt} \geq w(i,j)/2$. Depending on whether $i < j$, there are two cases. Below we only prove the case $i < j$, and the other case is very similar.

First of all, we claim that $|C(x_i^*, x_j^*)| \geq (j-i) \cdot \delta$. Indeed, consider any $k \in [i+1, j]$. Since OPT is an optimal solution, $|x_{k-1}^* x_k^*| \geq \delta$ holds. Because $|x_{k-1}^* x_k^*| = \min\{|C(x_{k-1}^*, x_k^*)|, |C(x_k^*, x_{k-1}^*)|\}$, we obtain that $|C(x_{k-1}^*, x_k^*)| \geq \delta$. Since the order of the points of P in OPT is the same as that in F_0, we have $C(x_i^*, x_j^*) = \cup_{k=i+1}^{j} C(x_{k-1}^*, x_k^*)$ and $|C(x_i^*, x_j^*)| = \sum_{k=i+1}^{j} |C(x_{k-1}^*, x_k^*)| \geq (j-i) \cdot \delta$. The claim is thus proved.

In the sequel, we prove $d_{opt} \geq w(i,j)/2 = [(j-i) \cdot \delta - |C(x_i, x_j)|]/2$.

If $|C(x_i^*, x_j^*)| - |C(x_i, x_j)| \leq 0$, then since $|C(x_i^*, x_j^*)| \geq (j-i) \cdot \delta$, it holds that $|C(x_i, x_j)| \geq (j-i) \cdot \delta$. Hence, $w(i,j) \leq 0$, and it follows that $d_{opt} \geq w(i,j)/2$.

If $|C(x_i^*, x_j^*)| - |C(x_i, x_j)| > 0$, then the difference of $|C(x_i^*, x_j^*)|$ and $|C(x_i, x_j)|$ is due to the moving of p_i and p_j. Because the order the points of P in OPT is the same as that in F_0, the smallest moving distance of these two points happens when x_i and x_j move towards opposite directions (i.e., x_i moves clockwise and x_j moves counterclockwise) by the same distance $(|C(x_i^*, x_j^*)| - |C(x_i, x_j)|)/2$. Therefore, we obtain $\max\{|x_i x_i^*|, |x_j x_j^*|\} \geq (|C(x_i^*, x_j^*)| - |C(x_i, x_j)|)/2$. Since $d_{opt} \geq \max\{|x_i x_i^*|, |x_j x_j^*|\}$, it holds that $d_{opt} \geq (|C(x_i^*, x_j^*)| - |C(x_i, x_j)|)/2$. Finally, because $|C(x_i^*, x_j^*)| \geq (j-i) \cdot \delta$, we obtain $d_{opt} \geq w(i,j)/2$. □

Based on Lemma 3.3.1, we obtain the following lemma, which is analogous to Lemma 3.2.3 for the line version.

Lemma 3.3.2. *If there exist $i \neq j$ in $[1, n]$ and a feasible configuration F' in which each point $p_k \in P$ is at location x'_k such that $w(i,j) = \max_{1 \leq k \leq n} |C(x_k, x'_k)|$, then we can obtain an optimal solution by shifting every point of P in F' clockwise by distance $w(i,j)/2$.*

Proof. Let F'' denote the configuration obtained by shifting every point of P in F' clockwise by distance $w(i,j)/2$.

Consider any point $p_k \in P$. Let x''_k denote the location of x_k in F''. On the one hand, $|C(x_k, x'_k)| \leq w(i,j)$ since $w(i,j) = \max_{1 \leq k \leq n} |C(x_k, x'_k)|$. On the other hand, since the above shifting moves p_k from x'_k clockwise to x''_k by distance $w(i,j)/2 \leq |C|/2$ (recall that $w(i,j) \leq |C|$), it holds that either $|C(x_k, x''_k)| \leq w(i,j)/2$ or $|C(x''_k, x_k)| \leq w(i,j)/2$. Consequently, $|x_k x''_k| = \min\{|C(x_k, x''_k)|, |C(x''_k, x_k)|\} \leq w(i,j)/2$.

The above shows that $\max_{1 \leq k \leq n} |C(x_k, x''_k)| \leq w(i,j)/2$, i.e., the maximum moving distance of all points of P in F'' is no more than $w(i,j)/2$. By Lemma 3.3.1, F'' is an optimal solution. The lemma is thus proved. □

We call a feasible configuration that satisfies the condition in Lemma 3.3.2 a *canonical configuration*. In light of Lemma 3.3.2, to solve the problem in linear time, it is sufficient to find a canonical configuration in linear time, which is our focus below.

3.3.2 Computing a Canonical Configuration

In this section, we present a linear-time algorithm that can find a canonical configuration. Comparing with the original problem, now we only need to consider the counterclockwise movements.

Recall that the points $p_1, p_2, \ldots, p_n$ are ordered on C counterclockwise in the input configuration F_0. For convenience of discussion, we define coordinates for locations on C in the following way. We define x_1 as the origin with coordinate zero. For any other location $x \in C$, the coordinate of x is defined to be $|C(x_1, x)|$. Hence each location of C has a coordinate no greater than $|C|$.

Our algorithm has two rounds. In the first round, we will use the same approach as for the line version of the problem, and let F_1 denote the resulting configuration. However, the issue is that in F_1 the new location of p_n may be too close to p_1 or p_n may even "cross" p_1, which might make F_1 not feasible. If p_n does not cross p_1 and p_n is at least δ away from p_1 in F_1, then we will show that F_1 is a canonical configuration. Otherwise, we will proceed to the second round, which is to (starting from the configuration F_1) consider all points again from p_1 and use the same strategy to set the new locations of the points. We will show that the configuration F_2 obtained after the second round is a canonical configuration. The details are given below.

The first round

In the first round, we will move each point $p_i \in P$ from x_i along C counterclockwise to a new location x_i'. The way we set x_i' here is similar to that in the line version and the difference is that we have to take care of the cycle situation. Specifically, $x_1' = x_1$, i.e., p_1 does not move. For each $i \in [2, n]$, suppose we have already moved p_{i-1} to x_{i-1}', then we define x_i' as follows:

$$x_i' = \begin{cases} x_i & \text{if } x_i \geq x_{i-1}' + \delta \\ (x_{i-1}' + \delta) \mod |C| & \text{if } x_i < x_{i-1}' + \delta. \end{cases} \tag{3.1}$$

This finishes the first round of our algorithm. Denote by F_1 the resulting configuration.

Note that if $x_{i-1}' + \delta > |C|$, then since $x_i \leq |C|$, according to Equation (3.1), $x_i' = (x_{i-1}' + \delta) \mod |C|$, which is equal to $x_{i-1}' + \delta - |C|$; in this case, we say that the counterclockwise movement of p_i *crosses* the origin x_1.

Lemma 3.3.3. *If p_n does not cross x_1 $(= x_1')$ in the first round of the algorithm and $|C(x_n', x_1')| \geq \delta$, then F_1 is a canonical configuration.*

Proof. First of all, we show that F_1 is a feasible configuration, i.e., the distance between any two points of P in F_1 is at least δ. Consider any two indices i and j. Without loss of generality, assume $i < j$. Our goal is to show that $|x_i'x_j'| \geq \delta$. To this end, it is sufficient to show that $|C(x_i', x_j')| \geq \delta$ and $|C(x_j', x_i')| \geq \delta$.

On the one hand, $C(x_i', x_j')$ contains x_{i+1}', implying that $C(x_i', x_{i+1}') \subseteq C(x_i', x_j')$ and thus $|C(x_i', x_{i+1}')| \leq |C(x_i', x_j')|$. According to our first round algorithm (i.e., Equation (3.1)), it holds that $|C(x_i', x_{i+1}')| \geq \delta$. Thus, $|C(x_i', x_j')| \geq \delta$.

On the other hand, since p_n does not cross $x_1 = x_1'$, $C(x_j', x_i')$ contains both x_n' and x_1', and in other words, $C(x_n', x_1') \subseteq C(x_j', x_i')$. Due to $|C(x_n', x_1')| \geq \delta$, we obtain $|C(x_j', x_i')| \geq |C(x_n', x_1')| \geq \delta$.

Therefore, F_1 is a feasible configuration.

Let d_{max}' be the maximum counterclockwise movement of all points of P in the first round, i.e., $d_{max}' = \max_{1 \leq k \leq n} |C(x_k, x_k')|$. To show that F_1 is canonical configuration, we also need to show that there exist i and j such that $d_{max}' = w(i, j)$. In the following, we will find two indices i and j with $i < j$ such that $d'_{max} = w(i,j)$. Recall that when $i < j$, $w(i,j) = (j-i) \cdot \delta - |C(x_i, x_j)|$.

Since the input configuration F_0 is not feasible, it must hold that $d'_{max} > 0$. Let j be the index such that $d'_{max} = |C(x_j, x'_j)|$. Let i be the largest index such that $i < j$ and $x'_i = x_i$. Note that such an index i must exist since $x_1 = x'_1$.

According to the definition of i, each point x_k with $i+1 \leq k \leq j$ is moved in the first round algorithm, which implies that $|C(x'_{k-1}, x'_k)| = \delta$ according to Equation (3.1). Hence, we obtain $|C(x'_i, x'_j)| = \sum_{k=i+1}^{j} |C(x'_{k-1}, x'_k)| = (j-i) \cdot \delta$. On the other hand, since the movement of p_n does not cross x_1 and p_i does not move, the movement of p_j does not cross $x_i = x'_i$. Thus, $C(x'_i, x'_j) = C(x_i, x_j) \cup C(x_j, x'_j)$ and $|C(x'_i, x'_j)| = |C(x_i, x_j)| + |C(x_j, x'_j)|$.

Therefore, we obtain $d'_{max} = |C(x_j, x'_j)| = |C(x'_i, x'_j)| - |C(x_i, x_j)| = (j-i) \cdot \delta - |C(x_i, x_j)| = w(i,j)$.

We conclude that F_1 is a canonical configuration. □

According to Lemma 3.3.3, if p_n does not cross $x_1 = x'_1$ in the first round and $|C(x'_n, x'_1)| \geq \delta$ in F_1, then we have found a canonical configuration and our algorithm stops. Otherwise, we proceed to the second round, as follows.

The second round

In the second round, we will move each point $p_i \in P$ from x'_i counterclockwise to a new location x''_i, defined as follows.

We first define x''_1. Recall that we proceed to the second round because either p_n crosses $x_1 = x'_1$ in the first round or $|C(x'_n, x'_1)| < \delta$. In either case we define

$$x''_1 = (x'_n + \delta) \mod |C|. \tag{3.2}$$

Hence, $|C(x'_n, x''_1)| = \delta$.

For each $i = 2, 3, \ldots, n$, suppose p_{i-1} has been moved to x''_{i-1}; then we move p_i from x'_i counterclockwise to x''_i, with

$$x''_i = \max\{x'_i, (x''_{i-1} + \delta) \mod |C|\} \tag{3.3}$$

This finishes the second round of our algorithm. Let F_2 be the resulting configuration. In the sequel we show that F_2 is a canonical configuration.

Recall that $|C| \geq n \cdot \delta$. We first have the following observation on the first round of the algorithm.

Observation 3.3.4. *There must be a point p_i with $i \in [2, n]$ such that p_i does not move in the first round of the algorithm (i.e., $x_i = x'_i$).*

Proof. Assume to the contrary that every point p_i with $i \in [2, n]$ is moved in the first round. Then, by our first round algorithm (i.e., Equation (3.1)), $|C(x'_{i-1}, x'_i)| = \delta$ for each $2 \leq i \leq n$. Hence, $|C(x'_1, x'_n)| = \sum_{i=2}^{n} |C(x'_{i-1}x'_i)| = (n-1) \cdot \delta$. Further, since either p_n crosses $x_1 = x'_1$ or $|C(x'_n, x'_1)| < \delta$, we obtain that $n \cdot \delta > |C|$, which contradicts with the fact that $|C| \geq n \cdot \delta$. □

Observation 3.3.5. *If a point p_i does not move in the second round, then for each point p_j with $j \in [i, n]$, p_j does not move in the second round either.*

Proof. If $i = n$, then the observation trivially follows. We assume $i < n$.

According to the first round algorithm, it holds that $|C(x'_{k-1}, x'_k)| \geq \delta$ for any $k \in [2, n]$. Since p_i does not move in the second round, $x''_i = x'_i$ holds. Due to $|C(x'_i, x'_{i+1})| \geq \delta$, according to our second round algorithm (e.g., Equation (3.3)), $x''_{i+1} = x'_{i+1}$. By the same reasoning, $x''_j = x'_j$ for any $j \in [i+1, n]$, which leads to the observation. □

With Observations 3.3.4 and 3.3.5, we can prove the following lemma.

Lemma 3.3.6. *Suppose k is the largest index such that p_k does not move in the first round of the algorithm; then p_k does not move in the second round of the algorithm either, i.e., $x_k = x'_k = x''_k$.*

Proof. According to the first round algorithm, it holds that $|C(x'_{i-1}, x'_i)| \geq \delta$ for any $i \in [2, n]$.

By Observation 3.3.4, $k \in [2, n]$. We first discuss the case where $k \in [3, n-1]$. Indeed, this is the most general case. As shown later, the case where $k = 2$ or $k = n$ can by proved by similar but simpler techniques.

By the definition of k, the points $p_{k+1}, p_{k+2}, \ldots, p_n$ are moved in the first round. Hence, for each $i \in [k+1, n]$, according to our first round algorithm (i.e., Equation (3.1)), $|C(x'_{i-1}, x'_i)| = \delta$. Thus,

$$|C(x'_k, x'_n)| = \sum_{i=k+1}^{n} |C(x'_{i-1}, x'_i)| = (n-k) \cdot \delta. \tag{3.4}$$

Recall that p_1 is moved in the second round, and according to Equation (3.2),

$$|C(x'_n, x''_1)| = \delta. \tag{3.5}$$

If there is any $i \in [2, k-1]$ such that p_i does not move in the second round, then by Observation 3.3.5, p_k does not move in the second round either, which leads to the lemma.

Otherwise, since every point p_i with $i \in [2, k-1]$ is moved in the second round, according to our second round algorithm (i.e., Equation (3.3)), $|C(x''_{i-1}, x''_i)| = \delta$ holds. Hence, we obtain

$$|C(x''_1, x''_{k-1})| = \sum_{i=2}^{k-1} |C(x''_{i-1}, x''_i)| = (k-2) \cdot \delta. \tag{3.6}$$

Based on Equations (3.4), (3.5), and (3.6), we obtain $|C(x'_k, x'_n)| + |C(x'_n, x''_1)| + |C(x''_1, x''_{k-1})| = (n-1) \cdot \delta$. This implies that in the second round the counterclockwise movement of p_{k-1} from x'_{k-1} to x''_{k-1} does not cross $x_k = x'_k$, due to $|C| \geq n \cdot \delta$. Further, $|C(x''_{k-1}, x'_k)| = |C| - |C(x'_k, x''_{k-1})| = |C| - (|C(x'_k, x'_n)| + |C(x'_n, x''_1)| + |C(x''_1, x''_{k-1})|) = |C| - (n-1) \cdot \delta \geq \delta$. According to our second round algorithm (i.e., Equation (3.3)), $x''_k = x'_k$, i.e., p_k does not move in the second round.

The above proves the lemma for the case where $k \in [3, n-1]$.

If $k = 2$ or $k = n$, the proof is very similar.

If $k = 2$, then we still have Equations (3.4) and (3.5). Thus, $|C(x'_2, x'_n)| + |C(x'_n, x''_1)| = (n-1) \cdot \delta$. This implies that in the second round the counterclockwise movement of p_1 from x'_1 to x''_1 does not cross $x_2 = x'_2$, due to $|C| \geq n \cdot \delta$. Further, $|C(x''_1, x'_2)| = |C| - |C(x'_2, x''_1)| = |C| - (|C(x'_2, x'_n)| + |C(x'_n, x''_1)|) = |C| - (n-1) \cdot \delta \geq \delta$. According to our second round algorithm (i.e., Equation (3.3)), $x''_2 = x'_2$, i.e., p_2 does not move in the second round. Hence, the lemma is proved.

If $k = n$, then we still have Equations (3.5) and (3.6). Thus, $|C(x'_n, x''_1)| + |C(x''_1, x''_{n-1})| = (n-1)\cdot\delta$. This implies that in the second round when p_{n-1} moved from x'_{n-1} to x''_{n-1}, p_{n-1} does not cross $x_n = x'_n$, due to $|C| \geq n \cdot \delta$. Further, $|C(x''_{n-1}, x'_n)| = |C| - |C(x'_n, x''_{n-1})| = |C| - (|C(x'_n, x''_1)| + |C(x''_1, x''_{n-1})|) = |C| - (n-1)\cdot\delta \geq \delta$. According to our second round algorithm (i.e., Equation (3.3)), $x'_n = x''_n$, i.e., p_n does not move in the second round. Hence, the lemma follows.

In summary, p_k does not move in the second round of the algorithm. □

Recall that F_2 is the configuration after the second round of the algorithm. Our goal is to prove that F_2 is a canonical configuration. Based on the proof of Lemma 3.3.6, we have the following two corollaries.

Corollary 3.3.7. *The configuration F_2 is feasible.*

Proof. Suppose p_k is the point specified in Lemma 3.3.6. Hence, $k \in [2, n]$ and p_k does not move in the two rounds of our algorithm. We only prove the case where $k \in [2, n-1]$, and the case $k = n$ can be proved by similar (but simpler) techniques.

After the first round, it holds that $|C(x'_{i-1}, x'_i)| \geq \delta$ for each $i \in [k+1, n]$. Since x_k does not move in the second round, by Observation 3.3.5, $x''_i = x'_i$ for any $i \in [k, n]$. Hence, for each $i \in [k+1, n]$, it holds that $|C(x''_{i-1}, x''_i)| \geq \delta$.

On the other hand, according to our second round algorithm, $|C(x'_n, x''_1)| \geq \delta$ and $|C(x''_{i-1}, x''_i)| \geq \delta$ for each $i \in [2, k]$. Since $x'_n = x''_n$, it holds that $|C(x''_n, x''_1)| = |C(x'_n, x''_1)| \geq \delta$.

The above discussion leads to the following *observation*: $x''_1, x''_2, \ldots, x''_n$ are ordered counterclockwise on C, and further, for each $i \in [2, n]$, $|C(x''_{i-1}, x''_i)| \geq \delta$, and $|C(x''_n, x''_1)| \geq \delta$.

To show that F_2 is feasible, our goal is to prove that $|x''_i x''_j| \geq \delta$ for any $i \neq j \in [1, n]$. Consider any $i \neq j \in [1, n]$. Without loss of generality, we assume $i < j$. To prove $|x''_i x''_j| \geq \delta$, it is sufficient to show that $|C(x''_i, x''_j)| \geq \delta$ and $|C(x''_j, x''_i)| \geq \delta$.

The above observation implies that $C(x''_i, x''_{i+1}) \subseteq C(x''_i, x''_j)$ and $|C(x''_i, x''_j)| \geq |C(x''_i, x''_{i+1})| \geq \delta$. On the other hand, $C(x''_n, x''_1) \subseteq C(x''_j, x''_i)$. Since $|C(x''_n, x''_1)| \geq \delta$, we have $|C(x''_j, x''_i)| \geq |C(x''_n, x''_1)| \geq \delta$.

Therefore, $|x''_i x''_j| \geq \delta$ holds. The corollary thus follows. □

Corollary 3.3.8. *The total counterclockwise moving distance of each point of P in the two rounds of the algorithm is at most $|C| - \delta$, which implies that $|C(x_i, x''_i)| \leq |C| - \delta$ for each $1 \leq i \leq n$.*

Proof. By Lemma 3.3.6, suppose p_k does not move in the two rounds of our algorithm. For each other point p_i with $i \neq k$, since p_k does not move in the algorithm, the counterclockwise movement of p_i in the two rounds of the algorithm does not cross x_k. Further, as shown in the proof of Corollary 3.3.7, both $|C(x_k, x''_i)| \geq \delta$ and $|C(x''_i, x_k)| \geq \delta$ hold. Hence, the maximum counterclockwise movement of p_i in the two rounds is no more than $|C| - \delta$. The corollary follows. □

Finally, the next lemma shows that F_2 is a canonical configuration.

Lemma 3.3.9. *The configuration F_2 is a canonical configuration.*

Proof. Corollary 3.3.7 has already shown that F_2 is a feasible configuration. To prove the lemma, it is sufficient to prove that there exist i and j in $[1, n]$ such that $d_{max} = w(i, j)$, where $d_{max} = \max_{1 \leq k \leq n} |C(x_k, x''_k)|$.

Let j be the index such that $d_{max} = |C(x_j, x''_j)|$. We define another index i as follows. If $j = 1$, or $j > 1$ but all points of $p_1, p_2, \ldots, p_{j-1}$ are moved in the two rounds of the algorithm, let i be the largest index in $[j + 1, n]$ such that p_i does not move in the two rounds of the algorithm; otherwise (i.e., $j > 1$ and at least one point of $p_1, p_2, \ldots, p_{j-1}$ does not move in the two rounds of the algorithm), let i be the largest index in $[1, j - 1]$ such that p_i does not move in the two rounds of the algorithm. By Lemma 3.3.6, such an index i must exists. In the following, we prove that $d_{max} = w(i, j)$.

Depending on whether $i \in [1, j - 1]$ or $i \in [j + 1, n]$, there are two cases.

1. If $i \in [1, j - 1]$, then by the definition of i, all points $p_{i+1}, p_{i+2}, \ldots, p_j$ are moved in the algorithm. Since p_i does not move in the second round, by Observation 3.3.5, for each $k \in [i + 1, n]$, x_k does not move in the second round. This implies

that every point of $p_{i+1}, p_{i+2}, \ldots, p_j$ is moved in the first round of the algorithm. According to our first round algorithm, $|C(x'_{k-1}, x'_k)| = \delta$ for each $k \in [i+1, j]$. Hence, $|C(x_i, x''_j)| = |C(x''_i, x''_j)| = \sum_{k=i+1}^{j} |C(x''_{k-1}, x''_k)| = (j-i) \cdot \delta$ (because $x''_k = x'_k$ for each $k \in [i, n]$).

Since $i < j$ and $x_i = x'_i = x''_i$, $C(x_i, x''_j) = C(x_i, x_j) \cup C(x_j, x''_j)$. Thus, $|C(x_j, x''_j)| = |C(x_i, x''_j)| - |C(x_i, x_j)| = (j-i) \cdot \delta - |C(x_i, x_j)|$, which is equal to $w(i, j)$ since $i < j$.

Hence, the lemma is proved for this case.

2. If $i \in [j+1, n]$, we only discuss the general case where $i < n$. The special case where $i = n$ can be proved by similar (but simpler) techniques.

   Consider any point p_k with $k \in [i+1, n]$. Since p_i does not move in the two rounds of the algorithm, by Observation 3.3.5, p_k does not move in the second round. According to the definition of i, p_k is moved in the algorithm. Hence, p_k is moved in the first round. According to our first round algorithm (i.e., Equation (3.1)), $|C(x'_{k-1}, x'_k)| = \delta$. Further, since $x''_k = x'_k$, $|C(x''_{k-1}, x''_k)| = \delta$ holds. Therefore, $|C(x''_i, x''_n)| = \sum_{k=i+1}^{n} |C(x''_{k-1}, x''_k)| = (n-i) \cdot \delta$.

   Since p_1 is moved in the second round, by Equation (3.2), $|C(x'_n, x''_1)| = \delta$. We have shown above that p_k does not move in the second round for any $k \in [i+1, n]$. Hence, $x''_n = x'_n$ and $|C(x''_n, x''_1)| = \delta$.

   If $j = 1$, then $|C(x''_i, x''_1)| = |C(x''_i, x''_n)| + |C(x''_n, x''_1)| = (n+1-i) \cdot \delta$. Further, since p_i does not move in the algorithm (i.e., $x''_i = x'_i = x_i$), $d_{\max} = |C(x_1, x''_1)| = |C(x_i, x''_1)| - |C(x_i, x_1)| = (n+1-i) \cdot \delta - |C(x_i, x_1)|$, which is equal to $w(i, 1)$. The lemma thus follows.

   In the following, we discuss the case $j > 1$.

   Consider any point p_k with $k \in [2, j]$.

   We claim that p_k is moved in the second round (i.e., $x'_k \neq x''_k$). We prove the claim by induction. Indeed, by the definition of i, p_k is moved in the two rounds of the algorithm. Recall that p_1 is moved in the second round. For any $k \in [2, j]$, suppose p_{k-1} is moved in the second round. Assume to the contrary that p_k does not move in the second round. Then, p_k must be moved in the first round. According to our first round algorithm (i.e., Equation (3.1)), $|C(x'_{k-1}, x'_k)| = \delta$. Since p_{k-1} is moved in the second round, p_k must be moved as well.

In light of the above claim and according to our second round algorithm, $|C(x''_{k-1}, x''_k)| = \delta$ for each $k \in [2, j]$. Therefore, we derive $|C(x''_1, x''_j)| = \sum_{k=2}^{j} |C(x''_{k-1}, x''_k)| = (j-1) \cdot \delta$.

Based on the above discussions, $|C(x''_i, x''_j)| = |C(x''_i, x''_n)| + |C(x''_n, x''_1)| + |C(x''_1, x''_j)| = (n+j-i) \cdot \delta$. Since $x_i = x''_i$, $d_{\max} = |C(x_j, x''_j)| = |C(x_i, x''_j)| - |C(x_i, x_j)| = (n+j-i) \cdot \delta - |C(x_i, x_j)|$, which is equal to $w(i, j)$.

As a summary, F_2 is a canonical configuration. □

Clearly, both rounds of our algorithm run in $O(n)$ time. Combining Lemmas 3.3.2, 3.3.3, and 3.3.9, we have the following result.

Theorem 3.3.10. *The cycle version of the points-spreading problem is solvable in $O(n)$ time.*

Remark: One may verify that our algorithm for computing the canonical configuration F_2 essentially solves the following *one-direction case* of the cycle version problem: Move the points of P counterclockwise such that any pair of points of P are at least δ away from each other and the maximum counterclockwise moving distance of all points of P is minimized.

3.4 The Facility-Location Movement Problem

In this section, we present our linear-time algorithm for the facility-location movement problem. In this problem, we are given a set S of k "server" points and a set Q of n "client" points sorted on a line L, and the goal is to move all servers and clients on L such that each client co-locates with a server and the maximum moving distance of all servers and clients is minimized.

As shown by Dumitrescu and Jiang [34], the problem is equivalent to finding k intervals (i.e., line segments) on L such that each interval contains at least one server, each client is covered by at least one interval, and the maximum length of these intervals is minimized. In the following, we will focus on solving this *interval coverage* problem (also called *constrained k-center* problem in [34]).

Dumitrescu and Jiang [34] presented an $O((n+k)\log(n+k))$ time algorithm using dynamic programming. We discover a monotonicity property on their dynamic programming scheme, and consequently improve their algorithm to $O(n+k)$ time. Below, we first review the algorithm in [34] and then show our improvement.

3.4.1 Preliminaries

Without loss of generality, we assume L is the x-axis. For any two points p and q on L with p to the left of q, we use $[p, q]$ to denote the interval on L with left endpoint at p and right endpoint at q. An easy observation is that there exists an optimal solution consisting of k intervals in $\{[p, q] \mid p, q \in S \cup P\}$. For any two points p and q on L, let $d(p, q)$ denote the distance between them.

Let $S = \{s_1, s_2, \ldots, s_k\}$ be the set of servers sorted on L from left to right. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be the set of clients sorted on L from left to right. For ease of exposition, we assume no two points in $S \cup Q$ are at the same location.

The servers of S partition the clients of Q into $k+1$ subsets, defined as follows. For each $i \in [1, k-1]$, let Q_i be the subset of the clients of Q between s_i and s_{i+1} on L. In addition, we let Q_0 be the subset of the clients of Q to the left of s_1, and let Q_k be the subset of the clients of Q to the right of s_k. Since both S and Q are already given sorted, we can obtain the subsets $Q_0, Q_1, \ldots, Q_k$ in $O(n+k)$ time. In the following, for simplicity of discussion, we assume Q_i is not empty for each $i \in [0, k]$. This implies that the rightmost client q_n is to the right of the rightmost server s_k and the leftmost client q_1 is to the left of the leftmost server s_1. For each $i \in [1, k]$, let $Q_i' = \{s_i\} \cup Q_i$.

3.4.2 A Dynamic Programming Algorithm

Consider any Q_i' with $1 \leq i \leq k$. Let q be any point in Q_i'. Consider the *subproblem* at q: Finding i intervals on L such that each interval contains at least one server of $\{s_1, s_2, \ldots, s_i\}$, each client to the left of q (including q if $q \neq s_i$) must be covered by at least one interval, and the maximum length of these i intervals is minimized. Define $\alpha(q)$ as the maximum length of the intervals in an optimal solution of the above subproblem at q. Our goal for the interval coverage problem is to solve the subproblem at q_n and compute the value $\alpha(q_n)$.

For any point $q \in S \cup Q$, we use $r(q)$ to denote right neighboring point of q on L in $S \cup Q$ (i.e., the closest point of $S \cup Q$ to q strictly to the right of q). Note that after merging S and Q into one sorted list, we can obtain $r(q)$ for each $q \in S \cup Q$ in constant time.

Initially, for each $q \in Q'_1$, $\alpha(q) = d(q_1, q)$ (recall that q_1 is to the left of s_1).

In general, consider any $q \in Q'_i$ for any $2 \leq i \leq k$. It holds that

$$\alpha(q) = \min_{q' \in Q'_{i-1}} \max\{\alpha(q'), d(r(q'), q)\}.$$

In words, in order to solve the subproblem at q, we use the $i-1$ intervals for the subproblem at q' along with an additional interval $[r(q'), q]$. To compute $\alpha(q)$, Dumitrescu and Jiang [34] used the following observation: As we consider the points q' of Q'_{i-1} from left to right, $\alpha(q')$ is monotonically increasing and $d(r(q'), q)$ is monotonically decreasing. Hence, if $\alpha(q')$ for all $q' \in Q'_{i-1}$ are known, $\alpha(q)$ can be computed in $O(\log |Q'_{i-1}|)$ time by binary search.

In this way, the value $\alpha(q_n)$ can be computed in $O((n + k) \log(n + k))$ time (more precisely, $O((n + k) \log n)$ time) and an optimal solution can be found correspondingly.

3.4.3 An Improved Implementation

We give an $O(n + k)$ time implementation for the above dynamic programming scheme. To this end, we find a new monotonicity property in Lemma 3.4.1.

Consider any point $q \in Q'_i$ such that $r(q)$ is still in Q'_i. For any point $q' \in Q'_{i-1}$, define $f(q') = \max\{\alpha(q'), d(r(q'), q)\}$. Hence, $\alpha(q) = \min_{q' \in Q'_{i-1}} f(q')$. Let $g(q)$ be the point in Q'_{i-1} such that $\alpha(q) = f(g(q))$ (if there is more than one such point, we let $g(q)$ refer to the rightmost one).

Lemma 3.4.1. *Either $g(r(q)) = g(q)$ or $g(r(q))$ is strictly to the right of $g(q)$.*

Proof. We only give an "intuitive" proof. Recall that as we consider the points q' of Q'_{i-1} from left to right, $\alpha(q')$ is monotonically increasing and $d(r(q'), q)$ is monotonically decreasing. Intuitively, $g(q)$ corresponds to the intersection of the two functions $\alpha(q')$

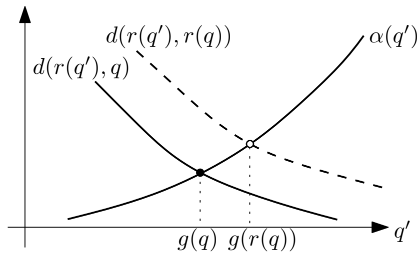


Figure 3.2. Illustrating the three functions $\alpha(q')$, $d(r(q'), q)$, and $d(r(q'), r(q))$ for $q' \in Q'_{i-1}$.

and $d(r(q'), q)$ for $q' \in Q'_{i-1}$ (e.g., see Fig. 3.2). Similarly, for the point $r(q)$, which is still in Q'_i, $g(r(q))$ corresponds to the intersection of the two functions $\alpha(q')$ and $d(r(q'), r(q))$ for $q' \in Q'_{i-1}$. An observation is that we can obtain the function $d(r(q'), r(q))$ by shifting $d(r(q'), q)$ upwards by the value $d(q, r(q))$ (e.g., see Fig. 3.2). This implies that $g(r(q))$ cannot be strictly to the left of $g(q)$. The lemma thus follows.

□

Lemma 3.4.1 essentially says that if we consider all points $q \in Q'_i$ from left to right, then $g(q)$ in Q'_{i-1} are also sorted on L from left to right. Due to this monotonicity property on $g(q)$, we can compute $g(q)$ and $\alpha(q)$ for all $q \in Q'_i$ in a total of $O(|Q'_{i-1}|+|Q'_i|)$ time by scanning the points of Q'_{i-1} from left to right. More specifically, suppose we have computed $g(q)$ and $\alpha(q)$ for some $q \in Q'_i$; then if $r(q)$ is still in Q'_i, we can compute $g(r(q))$ and $\alpha(r(q))$ by scanning the points of Q'_{i-1} starting from $g(q)$ to the right.

In this way, the value $\alpha(q_n)$ can be computed in $O(n + k)$ time, and an optimal solution can be found correspondingly. Hence, we have the following theorem.

Theorem 3.4.2. *If all servers and clients are sorted on the line L, then the facility-location movement problem can be solved in $O(n + k)$ time.*

As an application, our algorithm for Theorem 3.4.2 can be used to solve the cycle version of the same problem, where all servers and clients are given on a cycle. Dumitrescu and Jiang [34] showed that the cycle version can be solved by solving at most $(n + k)/k$ instances of the above line version of the problem (more specifically, there must be an adjacent pair of servers such that there are at most n/k clients between them; cutting the cycle between each adjacent pair of the above clients will result in an instance of the line version, with a total of no more than $(n+k)/k$ instances). By using their line-version algorithm of $O((n + k)\log(n + k))$ time, Dumitrescu and Jiang [34] solved the cycle version of the problem in $O(\frac{1}{k}(n+k)^2\log(n+k))$ time. By applying our improved algorithm for the line version, the cycle version can be solved in $O(\frac{1}{k}(n+k)^2)$ time.

CHAPTER 4

DISPERSING POINTS ON INTERVALS

4.1 Introduction

The problems of dispersing points have been extensively studied and can be classified to different categories by their different constraints and objectives, e.g., [41, 42, 45–48]. In this chapter, we consider problems of dispersing points on intervals in linear domains including lines and cycles. The results in this chapter have been published in a conference [20] and a journal [21].

4.1.1 Problem Definitions and Our Results

Let $\mathcal{I}$ be a set of n intervals on a line ℓ, and no two intervals of $\mathcal{I}$ intersect. The problem is to find a point in each interval of $\mathcal{I}$ such that the minimum distance of any pair of points is maximized. We assume the intervals of $\mathcal{I}$ are given sorted on ℓ. In this chapter we present an $O(n)$ time algorithm for this problem.

As an application of the problem, consider the following scenario. Suppose we are given n pairwise disjoint intervals on ℓ and we want to build a facility on each interval. As the facilities can interfere with each other if they are too close (e.g., if the facilities are hazardous), the goal is to choose locations for these facilities such that the minimum pairwise distance among these facilities is minimized. Clearly, this is an instance of our problem.

We also consider the *cycle version* of the problem where the intervals of $\mathcal{I}$ are given on a cycle $\mathcal{C}$. The intervals of $\mathcal{I}$ are also pairwise disjoint and are given sorted cyclically on $\mathcal{C}$. Note that the distance of two points on $\mathcal{C}$ is the length of the shorter arc of $\mathcal{C}$ between the two points. By making use of our "line version" algorithm, we solve this cycle version problem in linear time as well.

4.1.2 Related Work

To the best of our knowledge, we have not found any previous work on the two problems studied in this chapter. Our problems essentially belong to a family of geometric dispersion problems, which are NP-hard in general in two and higher dimensional space. For example, Baur and Fekete [49] studied the problems of distributing a number of points within a polygonal region such that the points are dispersed far away from each other, and they showed that the problems cannot be approximated arbitrarily well in polynomial time, unless P=NP.

Wang and Kuo [42] considered the following two problems. Given a set S of points and a value d, find a largest subset of S in which the distance of any two points is at least d. Given a set S of points and an integer k, find a subset of k points of S to maximize the minimum distance of all pairs of points in the subset. It was shown in [42] that both problems in 2D are NP-hard but can be solved efficiently in 1D. Refer to [50–54] for other geometric dispersion problems. Dispersion problems in various non-geometric settings were also considered [41, 45–48]. These problems are in general NP-hard; approximation and heuristic algorithms were proposed for them.

On the other hand, problems on intervals usually have applications in other areas. For example, some problems on intervals are related to scheduling because the time period between the release time and the deadline of a job or task in scheduling problems can be considered as an interval on the line. From the interval point of view, Garey et al. [6] studied the following problem on intervals: Given n intervals on a line, determine whether it is possible to find a unit-length sub-interval in each input interval, such that these sub-intervals do not intersect. An $O(n \log n)$ time algorithm was given in [6] for this problem. The optimization version of the above problem was also studied [55, 56], where the goal is to find a maximum number of intervals that contain non-intersecting unit-length sub-intervals. Chrobak et al. [55] gave an $O(n^5)$ time algorithm for the problem, and later Vakhania [56] improved the algorithm to $O(n^2 \log n)$ time. The online version of the problem was also considered [5]. Other optimization problems on intervals have also been considered, e.g., see [6, 8, 10, 11].

4.1.3 Our Approach

For the line version of the problem, our algorithm is based on a greedy strategy. We consider the intervals of $\mathcal{I}$ incrementally from left to right, and for each interval, we will "temporarily" determine a point in the interval. During the algorithm, we maintain a value $d_{\min}$, which is the minimum pairwise distance of the "temporary" points that so far have been computed. Initially, we put a point at the left endpoint of the first interval and set $d_{\min} = \infty$. During the algorithm, the value $d_{\min}$ will be monotonically decreasing. In general, when the next interval is considered, if it is possible to put a point in the interval without decreasing $d_{\min}$, then we put such a point as far left as possible. Otherwise, we put a point on the right endpoint of the interval. In the latter case, we also need to adjust the points that have been determined temporarily in the previous intervals that have been considered. We adjust these points in a greedy way such that $d_{\min}$ decreases the least. A straightforward implementation of this approach can only give an $O(n^2)$ time algorithm. In order to achieve the $O(n)$ time performance, during the algorithm we maintain a "critical list" $\mathscr{L}$ of intervals, which is a subset of intervals that have been considered. This list has some properties that help us implement the algorithm in $O(n)$ time.

We should point out that our algorithm is fairly simple and easy to implement. In contrast, the rationale of the idea is quite involved and it is not an easy task to argue its correctness. Indeed, discovering the critical list is the most challenging work and it is the key idea for solving the problem in linear time.

To solve the cycle version, the main idea is to convert the problem to a problem instance on a line and then apply our line version algorithm. More specifically, we make two copies of the intervals of $\mathcal{I}$ to a line and then apply our line version algorithm on these $2n$ intervals on the line. The line version algorithm will find $2n$ points in these intervals and we show that a particular subset of n consecutive points of them correspond to an optimal solution for the original problem on $\mathcal{C}$.

In the following, we will present our algorithms for the line version in Section 4.2. The cycle version is discussed in Section 4.3. Section 4.4 concludes.

4.2 The Line Version

Let $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ be the set of intervals sorted from left to right on ℓ. For any two points of p and q on ℓ, we use $|pq|$ to denote their distance. Our goal is to find a point p_i in I_i for each $1 \leq i \leq n$, such that the minimum pairwise distance of these points, i.e., $\min_{1 \leq i < j \leq n} |p_i p_j|$, is maximized.

For each interval I_i, $1 \leq i \leq n$, we use l_i and r_i to denote its left and right endpoints, respectively. We assume ℓ is the x-axis. With a little abuse of notation, for any point $p \in \ell$, depending on the context, p may also refer to its coordinate on ℓ. Therefore, for each $1 \leq i \leq n$, it is required that $l_i \leq p_i \leq r_i$.

For simplicity of discussion, we make a general position assumption that no two endpoints of the intervals of $\mathcal{I}$ have the same location (our algorithm can be easily extended to the general case). Note that this implies $l_i < r_i$ for any interval I_i.

The rest of this section is organized as follows. In Section 4.2.1, we discuss some observations. In Section 4.2.2, we give an overview of our algorithm. The details of the algorithm are presented in Section 4.2.3. Finally, we discuss the correctness and analyze the running time in Section 4.2.4.

4.2.1 Observations

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of sought points. Since all intervals are disjoint, $p_1 < p_2 < \ldots < p_n$. Note that the minimum pairwise distance of the points of P is also the minimum distance of all pairs of adjacent points.

Denote by d_{opt} the minimum pairwise distance of P in an optimal solution, and d_{opt} is called the *optimal objective value.* We have the following lemma.

Lemma 4.2.1. *$d_{opt} \leq \frac{r_j - l_i}{j - i}$ for any $1 \leq i < j \leq n$.*

Proof. Assume to the contrary that this is not true. Then there exist i and j with $i < j$ such that $d_{opt} > \frac{r_j - l_i}{j - i}$. Consider any optimal solution OPT. Note that in OPT, $p_i, p_{i+1}, \ldots, p_j$ are located in the intervals $I_i, I_{i+1}, \ldots, I_j$, respectively, and $|p_i p_j| \geq d_{opt} \cdot (j - i)$. Hence, $|p_i p_j| > r_j - l_i$. On the other hand, since $l_i \leq p_i$ and $p_j \leq r_j$, it holds that $|p_i p_j| \leq r_j - l_i$. We thus obtain contradiction. □

The preceding lemma leads to the following corollary.

Corollary 4.2.2. *Suppose we find a solution (i.e., a way to place the points of P) in which the minimum pairwise distance of P is equal to $\frac{r_j - l_i}{j-i}$ for some $1 \le i < j \le n$. Then the solution is an optimal solution.*

Our algorithm will find such a solution as stated in the corollary.

4.2.2 The Algorithm Overview

Our algorithm will consider and process the intervals of $\mathcal{I}$ one by one from left to right. Whenever an interval I_i is processed, we will "temporarily" determine p_i in I_i. We say "temporarily" because later the algorithm may change the location of p_i. During the algorithm, a value $d_{\min}$ and two indices i^* and j^* will be maintained such that $d_{\min} = (r_{j^*} - l_{i^*})/(j^* - i^*)$ always holds.

Initially, we set $p_1 = l_1$ and $d_{\min} = \infty$, with $i^* = j^* = 1$. In general, suppose the first $i-1$ intervals have been processed; then $d_{\min}$ is equal to the minimum pairwise distance of the points $p_1, p_2, \ldots, p_{i-1}$, which have been temporarily determined. In fact, $d_{\min}$ is the optimal objective value for the sub-problem on the first $i-1$ intervals. During the execution of algorithm, $d_{\min}$ will be monotonically decreasing. After all intervals are processed, $d_{\min}$ is d_{opt}. When we process the next interval I_i, we temporarily determine p_i in a greedy manner as follows. If $p_{i-1} + d_{\min} \le l_i$, we put p_i at l_i. If $l_i < p_{i-1} + d_{\min} \le r_i$, we put p_i at $p_{i-1} + d_{\min}$. If $p_{i-1} + d_{\min} > r_i$, we put p_i at r_i. In the first two cases, $d_{\min}$ does not change. In the third case, however, $d_{\min}$ will decrease. Further, in the third case, in order to make the decrease of $d_{\min}$ as small as possible, we need to move some points of $\{p_1, p_2, \ldots, p_{i-1}\}$ leftwards. By a straightforward approach, this moving procedure can be done in $O(n)$ time. But this will make the entire algorithm run in $O(n^2)$ time.

To have any hope of obtaining an $O(n)$ time algorithm, we need to perform the above moving "implicitly" in $O(1)$ amortized time. To this end, we need to find a way to answer the following question: Which points of $p_1, p_2, \ldots, p_{i-1}$ should move leftwards and how far should they move? To answer the question, the crux of our algorithm is to maintain a "critical list" $\mathscr{L}$ of interval indices, which bears some important properties that eventually help us implement our algorithm in $O(n)$ time.

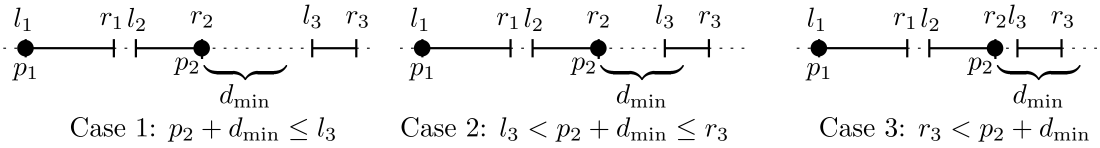


Figure 4.1. Illustrating the three cases when I_3 is being processed.

In fact, our algorithm is fairly simple. The most "complicated" part is to use a linked list to store $\mathscr{L}$ so that the following three operations on $\mathscr{L}$ can be performed in constant time each: remove the front element; remove the rear element; add a new element to the rear. Refer to Algorithm 1 for the pseudocode.

Although the algorithm is simple, the rationale of the idea is rather involved and it is also not obvious to see the correctness. Indeed, discovering the critical list is the most challenging task and the key idea for designing our linear time algorithm. To help in understanding and give some intuition, below we use an example of only three intervals to illustrate how the algorithm works.

Initially, we set $p_1 = l_1$, $d_{\min} = \infty$, $i^* = j^* = 1$, and $\mathscr{L} = \{1\}$.

To process I_2, we first try to put p_2 at $p_1 + d_{\min}$. Clearly, $p_1 + d_{\min} > r_2$. Hence, we put p_2 at r_2. Since p_1 is already at l_1, which is the leftmost point of I_1, we do not need to move it. We update $j^* = 2$ and $d_{\min} = r_2 - l_1$. Finally, we add 2 to the rear of $\mathscr{L}$. This finishes the processing of I_2.

Next we process I_3. We try to put p_3 at $p_2 + d_{\min}$. Depending on whether $p_2 + d_{\min}$ is to the left of I_3, in I_3, or to the right of I_3, there are three cases (e.g., see Fig. 4.1).

1. If $p_2 + d_{\min} \leq l_3$, we set $p_3 = l_3$. We reset $\mathscr{L}$ to $\{3\}$. None of $d_{\min}$, i^*, and j^* needs to be changed in this case.

2. If $l_3 < p_2 + d_{\min} \leq r_3$, we set $p_3 = p_2 + d_{\min}$. None of $d_{\min}$, i^*, and j^* needs to be changed. Further, the critical list $\mathscr{L}$ is updated as follows.

   We first give some "motivation" on why we need to update $\mathscr{L}$. Assume later in the algorithm, say, when we process the next interval, we need to move both p_2 and p_3 leftwards simultaneously so that $|p_1p_2| = |p_2p_3|$ during the moving (this is for making $d_{\min}$ as large as possible). The moving procedure stops once either p_2

arrives at l_2 or p_3 arrives at l_3. To determine which case happens first, it suffices to determine whether $l_2 - l_1 > \frac{l_3 - l_1}{2}$.

(a) If $l_2 - l_1 > \frac{l_3 - l_1}{2}$, then p_2 will arrive at l_2 first, after which p_2 cannot move leftwards any more in the rest of the algorithm but p_3 can still move leftwards.

(b) Otherwise, p_3 will arrive at l_3 first, after which p_3 cannot move leftwards any more. However, although p_2 can still move leftwards, doing that would not help in making $d_{\min}$ larger.

We therefore update $\mathscr{L}$ as follows. If $l_2 - l_1 > \frac{l_3 - l_1}{2}$, we add 3 to the rear of $\mathscr{L}$. Otherwise, we first remove 2 from the rear of $\mathscr{L}$ and then add 3 to the rear.

3. If $r_3 < p_2 + d_{\min}$, we set $p_3 = r_3$. Since $|p_2p_3| < d_{\min}$, $d_{\min}$ needs to be decreased. To make $d_{\min}$ as large as possible, we will move p_2 leftwards until either $|p_1p_2|$ becomes equal to $|p_2p_3|$ or p_2 arrives at l_2. To determine which event happens first, we only need to check whether $l_2 - l_1 > \frac{r_3 - l_1}{2}$.

   (a) If $l_2 - l_1 > \frac{r_3 - l_1}{2}$, the latter event happens first. We set $p_2 = l_2$ and update $d_{\min} = r_3 - l_2$ $(= |p_2p_3|)$, $i^* = 2$, and $j^* = 3$. Finally, we remove 1 from the front of $\mathscr{L}$ and add 3 to the rear of $\mathscr{L}$, after which $\mathscr{L} = \{2, 3\}$.

   (b) Otherwise, the former event happens first. We set $p_2 = l_1 + \frac{r_3 - l_1}{2}$ and update $d_{\min} = (r_3 - l_1)/2$ $(= |p_1p_2| = |p_2p_3|)$ and $j^* = 3$ (i^* is still 1). Finally, we update $\mathscr{L}$ in the same way as the above second case. Namely, if $l_2 - l_1 > \frac{l_3 - l_1}{2}$, we add 3 to the rear of $\mathscr{L}$; otherwise, we remove 2 from $\mathscr{L}$ and add 3 to the rear.

One may verify that in any case the above obtained $d_{\min}$ is an optimal objective value for the three intervals.

As another example, Fig. 4.2 illustrates the solution found by our algorithm on six intervals.

4.2.3 The Algorithm

We are ready to present the details of our algorithm. For any two indices $i < j$, let $P(i, j) = \{p_i, p_{i+1}, \ldots, p_j\}$.

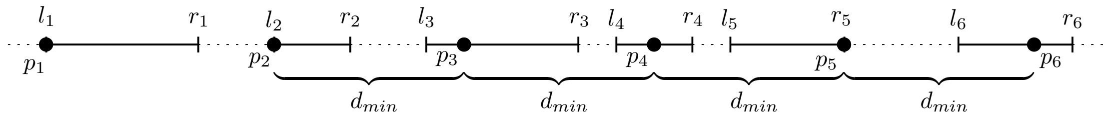


Figure 4.2. Illustrating the solution computed by our algorithm, with $i^* = 2$ and $j^* = 5$.

Initially we set $p_1 = l_1$, $d_{\min} = \infty$, $i^* = j^* = 1$, and $\mathscr{L} = \{1\}$. Suppose interval $i-1$ has just been processed for some $i > 1$. Let the current critical list be $\mathscr{L} = \{k_s, k_{s+1}, \dots k_t\}$ with $1 \le k_s < k_{s+1} < \cdots < k_t \le i-1$, i.e., $\mathscr{L}$ consists of $t-s+1$ sorted indices in $[1, i-1]$. Our algorithm maintains the following *invariants*.

1. The "temporary" location of p_{i-1} is known.

2. $d_{\min} = (r_{j^*} - l_{i^*})/(j^* - i^*)$ with $1 \le i^* \le j^* \le i-1$.

3. $k_t = i-1$.

4. $p_{k_s} = l_{k_s}$, i.e., p_{k_s} is at the left endpoint of the interval I_{k_s}.

5. The locations of all points of $P(1, k_s)$ have been explicitly computed and *finalized* (i.e., they will never be changed in the later algorithm).

6. For each $1 \le j \le k_s$, p_j is in I_j.

7. The distance of every pair of adjacent points of $P(1, k_s)$ is at least $d_{\min}$.

8. For each j with $k_s + 1 \le j \le i-1$, p_j is "implicitly" set to $l_{k_s} + d_{\min} \cdot (j - k_s)$ and $p_j \in I_j$. In other words, the distance of every pair of adjacent points of $P(k_s, i-1)$ is exactly $d_{\min}$.

9. The critical list $\mathscr{L}$ has the following *priority property*: If $\mathscr{L}$ has more than one element (i.e., $s < t$), then for any h with $s \le h \le t-1$, Inequality (4.1) holds for any j with $k_h + 1 \le j \le i-1$ and $j \ne k_{h+1}$.

$$\frac{l_{k_{h+1}} - l_{k_h}}{k_{h+1} - k_h} > \frac{l_j - l_{k_h}}{j - k_h}. \tag{4.1}$$

We give some intuition on what the priority property implies. Suppose we move all points in $P(k_s+1, i-1)$ leftwards simultaneously such that the distances between all adjacent pairs of points of $P(k_s, i-1)$ keep the same (by the above eighth invariant, they are the same before the moving). Then, Inequality (4.1) with $h = s$ implies that $p_{k_{s+1}}$ is the first point of $P(k_s+1, i-1)$ that arrives at the left endpoint of its interval. Once $p_{k_{s+1}}$ arrives at the interval left endpoint, suppose we continue to move the points of $P(k_{s+1}+1, i-1)$ leftwards simultaneously such that the distances between all adjacent pairs of points of $P(k_{s+1}, i-1)$ are the same. Then, Inequality (4.1) with $h = s+1$ makes sure that $p_{k_{s+2}}$ is the first point of $P(k_{s+1}+1, i-1)$ that arrives at the left endpoint of its interval. Continuing the above can explain the inequality for $h = s+2, s+3, \ldots, t-1$.

The priority property further leads to the following observation.

Observation 4.2.3. *For any h with $s \leq h \leq t-2$, the following holds:*

$$\frac{l_{k_{h+1}} - l_{k_h}}{k_{h+1} - k_h} > \frac{l_{k_{h+2}} - l_{k_{h+1}}}{k_{h+2} - k_{h+1}}.$$

Proof. Note that $k_h + 1 \leq k_{h+1} < k_{h+2} \leq i-1$. Let $j = k_{h+2}$. By Inequality (4.1), we have

$$\frac{l_{k_{h+1}} - l_{k_h}}{k_{h+1} - k_h} > \frac{l_{k_{h+2}} - l_{k_h}}{k_{h+2} - k_h}. \tag{4.2}$$

Note that for any four positive numbers a, b, c, d such that $a < c$, $b < d$, and $\frac{a}{b} > \frac{c}{d}$, it holds that $\frac{a}{b} > \frac{c-a}{d-b}$. Applying this to Inequality (4.2) will obtain the observation. □

Remark.. By Corollary 4.2.2, Invariants (2), (6), (7), and (8) together imply that $d_{\min}$ is the optimal objective value for the sub-problem on the first $i-1$ intervals.

One may verify that initially after I_1 is processed, all invariants trivially hold (we finalize p_1 at l_1). In the following we describe the general step of our algorithm to process the interval I_i. We will also show that all algorithm invariants hold after I_i is processed.

Depending on whether $p_{i-1} + d_{\min}$ is to the left of I_i, in I_i, or to the right of I_i, there are three cases.

The case $p_{i-1} + d_{\min} \leq l_i$

In this case, $p_{i-1} + d_{\min}$ is to the left of I_i. We set $p_i = l_i$ and finalize it. We do not change $d_{\min}$, i^*, or j^*. Further, for each $j \in [k_s + 1, i - 1]$, we explicitly compute $p_j = l_{k_s} + d_{\min} \cdot (j - k_s)$ and finalize it. Finally, we reset $\mathscr{L} = \{i\}$.

Lemma 4.2.4. *In the case $p_{i-1} + d_{\min} \leq l_i$, all algorithm invariants hold after I_i is processed.*

Proof. Recall that $\mathscr{L} = \{i\}$ after I_i is processed. Hence, $k_s = k_t = i$. For the sake of differentiation, we use $\mathscr{L}' = \{k'_s, k'_{s+1}, \ldots, k'_{t'}\}$ to denote the critical list before we process I_i.

1. Since p_i is known, Invariant (1) hold.

2. For Invariant (2), since the same invariant holds before we process I_i and none of $d_{\min}$, i^*, and j^* is changed when we process I_i, Invariant (2) trivially holds after we process I_i.

3. Since $k_t = i$, the third invariant holds.

4. Recall that $p_{k_s} = p_i = l_i$, which is the fourth invariant.

5. To prove Invariant (5), since the same invariant holds before I_i is processed, it is sufficient to show that the points of $P(k'_s + 1, i)$ have been explicitly computed and finalized in the step of processing I_i, which is clearly true according to our algorithm.

6. To prove Invariant (6), since the same invariant holds before I_i is processed, it is sufficient to show that each point p_j of $P(k'_s + 1, i)$ is in I_j.

   Indeed, consider any $j \in [k'_s + 1, i]$. If $j = i$, then since $p_j = l_j$, it is true that p_j is in I_j. If $j < i$, then by Invariant (8) of $\mathscr{L}'$, $l_{k'_s} + d_{\min} \cdot (j - k'_s)$ is in I_j. According to our algorithm, in the step of processing I_i, p_j is explicitly set to $l_{k'_s} + d_{\min} \cdot (j - k'_s)$. Hence, p_j is in I_j.

7. To prove Invariant (7), since the same invariant holds before I_i is processed, it is sufficient to show that $|p_{i-1}p_i| \geq d_{\min}$, which is clearly true according to our algorithm.

8. Invariant (8) trivially holds since $k_s + 1 > i$ (i.e., there is no j such that $k_s + 1 \leq j \leq i$).

9. Invariant (9) also holds since $\mathscr{L}$ has only one element.

This proves that all algorithm invariants hold. The lemma thus follows. □

The case $l_i < p_{i-1} + d_{\min} \leq r_i$

In this case, $p_{i-1} + d_{\min}$ is in I_i. We set $p_i = p_{i-1} + d_{\min}$. We do not change $d_{\min}$, i^*, or j^*. We update the critical list $\mathscr{L}$ by the following *rear-processing procedure* (because the elements of $\mathscr{L}$ are considered from the rear to the front).

If $s = t$, i.e., $\mathscr{L}$ only has one element, then we simply add i to the rear of $\mathscr{L}$. Otherwise, we first check whether the following inequality is true.

$$\frac{l_{k_t} - l_{k_{t-1}}}{k_t - k_{t-1}} > \frac{l_i - l_{k_{t-1}}}{i - k_{t-1}}. \tag{4.3}$$

If it is true, then we add i to the end of $\mathscr{L}$.

If it is not true, then we remove k_t from $\mathscr{L}$ and decrease t by 1. Next, we continue to check whether Inequality (4.3) (with the decreased t) is true and follow the same procedure until either the inequality becomes true or $s = t$. In either case, we add i to the end of $\mathscr{L}$. Finally, we increase t by 1 to let k_t refer to i.

This finishes the rear-processing procedure for updating $\mathscr{L}$.

Lemma 4.2.5. *In the case $l_i < p_{i-1} + d_{\min} \leq r_i$, all algorithm invariants hold after I_i is processed.*

Proof. For the sake of differentiation, we use $\mathscr{L}' = \{k'_s, k'_{s+1}, \ldots, k'_{t'}\}$ to denote the critical list before we process I_i. After I_i is processed, we have $\mathscr{L} = \{k_s, k_{s+1}, \ldots, k_t\}$. According to our algorithm, $\mathscr{L}$ is obtained from $\mathscr{L}'$ by possibly removing some elements of $\mathscr{L}'$ from the rear and then adding i to the end. Hence, $k_h = k'_h$ for any $h \in [s, t-1]$ and $k_t = i$. In particular, $k_s = k'_s$ since $\mathscr{L}$ has at least two elements (i.e., $s < t$).

1. Since the "temporary" location of p_i is computed, the first invariant holds.

2. The second invariant trivially holds since none of $d_{\min}$, i^*, and j^* is changed when we process I_i.

3. Since $k_t = i$, Invariant (3) holds.

4. To prove Invariant (4), we need to show that $p_{k_s} = l_{k_s}$. Since the same invariant holds for $\mathscr{L}'$, $p_{k'_s} = l_{k'_s}$. Due to $k_s = k'_s$, we obtain $p_{k_s} = l_{k_s}$.

5. Invariant (5) trivially holds since $k_s = k'_s$ and the same invariant holds before I_i is processed.

6. Similarly, since $k_s = k'_s$, Invariant (6) holds.

7. Similarly, since $k_s = k'_s$, Invariant (7) holds.

8. To prove Invariant (8), we need to show that p_j is implicitly set to $l_{k_s} + d_{\min} \cdot (j - k_s)$ and $p_j \in I_j$ for each $j \in [k_s + 1, i]$.

   Recall that $k_s = k'_s$ and $d_{\min}$ does not change when we process I_i. Since the same invariant holds before I_j is processed, for $j \in [k_s + 1, i - 1]$, it is true that p_j is implicitly set to $l_{k_s} + d_{\min} \cdot (j - k_s)$ and $p_j \in I_j$. For $j = i$, since $p_i = p_{i-1} + d_{\min}$ and $p_i \in I_i$, $p_i = l_{k_s} + d_{\min} \cdot (i - k_s)$.

   Hence, this invariant also holds.

The above has proved that the first eight invariants hold. It remains to prove the last invariant, i.e., the priority property of $\mathscr{L}$. Our goal is to show that for any $h \in [s, t - 1]$, Inequality (4.1) holds for any $j \in [k_h + 1, i]$ with $j \neq k_{h+1}$.

Consider any $h \in [s, t - 1]$ and any $j \in [k_h + 1, i]$ with $j \neq k_{h+1}$. Since $h \leq t - 1$, $k'_h = k_h$. Depending on whether $h \leq t - 2$ or $h = t - 1$, there are two cases.

The case $h \leq t - 2$.. In this case, $h + 1 \leq t - 1$ and thus $k'_{h+1} = k_{h+1}$.

If $j \leq i - 1$, then $j \in [k_h + 1, i - 1] = [k'_h + 1, i - 1]$. Since the priority property holds for $\mathscr{L}'$, we have $\frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h} > \frac{l_j - l_{k'_h}}{j - k'_h}$. As $k'_h = k_h$ and $k'_{h+1} = k_{h+1}$, Inequality (4.1) hold for j and h.

If $j = i$, then Inequality (4.1) can be proved with the help of Observation 4.2.3, as follows.

Since $h \leq t - 2$ and $s \leq h < t - 1$, k_s is not k_{t-1}. Since k_{t-1} is not removed from $\mathscr{L}$, according to our algorithm, Inequality (4.3) must be true with replacing t by $t - 1$, i.e., $\frac{l_{k_{t-1}} - l_{k_{t-2}}}{k_{t-1} - k_{t-2}} > \frac{l_i - l_{k_{t-2}}}{i - k_{t-2}}$.

Further, recall that $k_m = k'_m$ for all $m \in [s, t-1]$. Due to the priority property of $\mathscr{L}'$ and by Observation 4.2.3, we obtain $\frac{l_{k_{h+1}} - l_{k_h}}{k_{h+1} - k_h} > \frac{l_{k_{t-1}} - l_{k_{t-2}}}{k_{t-1} - k_{t-2}}$.

Combining the above two inequalities gives us

$$\frac{l_{k_{h+1}} - l_{k_h}}{k_{h+1} - k_h} > \frac{l_i - l_{k_{t-2}}}{i - k_{t-2}}. \tag{4.4}$$

Depending on whether $h < t-2$, there are further two subcases.

1. If $h = t-2$, then Inequality (4.4) is Inequality (4.1) for $j = i$. So we are done with the proof.

2. If $h < t-2$, then, $k_h < k_{t-2} \le i-1$. Recall that $k'_h = k_h$ and $k'_{t-2} = k_{t-2}$. Due to the priority property of $\mathscr{L}'$ and by setting $j = k'_{t-2}$ in Inequality (4.1), we obtain $\frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h} > \frac{l_{k'_{t-2}} - l_{k'_h}}{k'_{t-2} - k'_h}$.

   Again, because $k'_h = k_h$, $k'_{h+1} = k_{h+1}$, and $k'_{t-2} = k_{t-2}$, we have

$$\frac{l_{k_{h+1}} - l_{k_h}}{k_{h+1} - k_h} > \frac{l_{k_{t-2}} - l_{k_h}}{k_{t-2} - k_h}. \tag{4.5}$$

   Note that for any positive numbers x, a, b, c, d such that $x > \frac{a}{b}$ and $x > \frac{c}{d}$, it always holds that $x > \frac{a+c}{b+d}$. Applying this to Inequalities (4.4) and (4.5) leads to $\frac{l_{k_{h+1}} - l_{k_h}}{k_{h+1} - k_h} > \frac{l_i - l_{k_h}}{i - k_h}$, which is Inequality (4.1) for $j = i$.

This proves Inequality (4.1) for the case $h \le t-2$.

The case $h = t-1$.. In this case, $k_{h+1} = k_t = i$. Due to $j \ne k_{h+1}$, $j \ne i$.

If none of the elements of $\mathscr{L}'$ was removed when we updated $\mathscr{L}$, i.e., $\mathscr{L} = \mathscr{L}' \cup \{i\}$, then $k_{t-1} = k'_{t'}$. Since $k'_{t'} = i-1$, $k_h = k_{t-1} = k'_{t'} = i-1$. Therefore, $k_h + 1 = i$, and there is no j with $k_h + 1 \le j \le i$ and $j \ne k_{h+1}$ $(= k_t = i)$. Hence, we have nothing to prove for Inequality (4.1) in this case.

In the following, we assume at least one element was removed from $\mathscr{L}'$ when we updated $\mathscr{L}$. Since $k'_{t-1} = k_{t-1}$ is the last element of $\mathscr{L}'$ remaining in $\mathscr{L}$, k'_t is the last element removed from $\mathscr{L}'$ when we process I_i. According to the algorithm, k'_t was removed because Inequality (4.3) was not true, i.e., the following holds

$$\frac{l_{k'_t} - l_{k'_{t-1}}}{k'_t - k'_{t-1}} \leq \frac{l_i - l_{k'_{t-1}}}{i - k'_{t-1}}. \tag{4.6}$$

Recall that $k_h + 1 \leq j \leq i$, $j \neq i$, and $k_h = k_{t-1} = k'_{t-1}$. Due to the priority property of $\mathscr{L}'$ and by setting $h = t - 1$ in Inequality (4.1), we obtain

$$\frac{l_{k'_t} - l_{k'_{t-1}}}{k'_t - k'_{t-1}} > \frac{l_j - l_{k'_{t-1}}}{j - k'_{t-1}}. \tag{4.7}$$

Combining Inequalities (4.6) and (4.7), we obtain $\frac{l_i - l_{k'_{t-1}}}{i - k'_{t-1}} > \frac{l_j - l_{k'_{t-1}}}{j - k'_{t-1}}$, which is Inequality (4.1) for h and j since $h = t - 1$, $k'_t = k_t = i$, and $k'_{t-1} = k_{t-1}$.

The above proves that the priority property holds for the updated list $\mathscr{L}$.

This proves that all algorithm invariants hold after I_i is processed. □

The case $p_{i-1} + d_{\min} > r_i$

In this case, $p_{i-1} + d_{\min}$ is to the right of I_i. We first set $p_i = r_i$. Then we perform the following *front-processing procedure* (because it processes the elements of $\mathscr{L}$ from the front to the rear).

If $\mathscr{L}$ has only one element (i.e., $s = t$), then we stop.

Otherwise, we check whether the following is true

$$\frac{l_{k_{s+1}} - l_{k_s}}{k_{s+1} - k_s} > \frac{r_i - l_{k_s}}{i - k_s}. \tag{4.8}$$

If it is true, then we perform the following *finalization step*: for each $j = k_s + 1, k_s + 2, \ldots, k_{s+1}$, we explicitly compute $p_j = l_{k_s} + \frac{l_{k_{s+1}} - l_{k_s}}{k_{s+1} - k_s} \cdot (j - k_s)$ and finalize it. Further, we remove k_s from $\mathscr{L}$ and increase s by 1. Next, we continue the same procedure as above (with the increased s), i.e., first check whether $s = t$, and if not, check whether Inequality (4.8) is true. The front-processing procedure stops if either $s = t$ (i.e., $\mathscr{L}$ only has one element) or Inequality (4.8) is not true.

After the front-processing procedure, we update $d_{\min} = (r_i - l_{k_s})/(i - k_s)$, $i^* = k_s$, and $j^* = i$. Finally, we update the critical list $\mathscr{L}$ using the rear-processing procedure, in the same way as in the above second case where $l_i < p_{i-1} + d_{\min} \leq r_i$. We also "implicitly" set $p_j = l_{k_s} + d_{\min} \cdot (j - k_s)$ for each $j \in [k_s + 1, i]$ (this is only for the analysis and our algorithm does not do so explicitly).

This finishes the processing of I_i.

Lemma 4.2.6. *In the case $p_{i-1} + d_{\min} > r_i$, all algorithm invariants hold after I_i is processed.*

Proof. Let $\mathscr{L} = \{k_s, k_{s+1}, \ldots, k_t\}$ be the critical list after I_i is processed. For the sake of differentiation, we use $\mathscr{L}' = \{k'_s, k'_{s+1}, \ldots, k'_{t'}\}$ to denote the critical list before we process I_i.

According to our algorithm, $\mathscr{L}$ is obtained from $\mathscr{L}'$ by the following two main steps: (1) the front-processing step that possibly removes some elements of $\mathscr{L}'$ from the front; (2) the rear-processing step that possibly removes some elements of $\mathscr{L}'$ from the rear and then adds i to the rear. Hence, $k_t = i$.

Let w be the index of $\mathscr{L}'$ such that $k_s = k'_w$. If $w \neq s$, then $k'_s, k'_{s+1}, \ldots, k'_{w-1}$ are not in $\mathscr{L}$.

The first invariant.. Since the "temporary" location of p_i is computed with $p_i = r_i$, the first invariant holds.

The second invariant.. By our way of updating $d_{\min}$, i^*, and j^*, it holds that $d_{\min} = (r_{j^*} - l_{i^*})/(j^* - i^*)$, with $1 \leq i^* \leq j^* \leq i$. Hence, the invariant holds.

The third invariant.. Since $k_t = i$, the third invariant trivially holds.

The fourth invariant.. We need to show that $p_{k_s} = l_{k_s}$.

If $s = w$, then $k_s = k'_s$ and k_s is also the first element of $\mathscr{L}'$. Since the fourth invariant holds before I_i is processed, $p_{k'_s} = l_{k'_s}$. Thus, we obtain $p_{k_s} = l_{k_s}$.

If $s \neq w$, then when k'_{w-1} was removed from $\mathscr{L}$ in the algorithm, the finalization step explicitly computed $p_j = l_{k'_{w-1}} + \frac{l_{k'_w} - l_{k'_{w-1}}}{k'_w - k'_{w-1}} \cdot (j - k'_{w-1})$ for each $j \in [k'_{w-1} + 1, k'_w]$. Once can verify that $p_{k'_w} = l_{k'_w}$. Since $k'_w = k_s$, we obtain $p_{k_s} = l_{k_s}$.

This proves that the fourth invariant also holds.

The fifth invariant.. Our goal is to show that all points in $P(1, k_s)$ have been finalized. Since all points in $P(1, k'_s)$ have been finalized before we process I_i, it is sufficient to show that the points for $P(k'_s+1, k_s)$ were finalized in the step of processing I_i.

If $w = s$, then $k_s = k'_s$ and we are done with the proof. Otherwise, for each $h \in [s, w-1]$, when k'_h was removed from $\mathscr{L}$, the finalization step finalized the points in $P(k'_h+1, k'_{h+1})$. Hence, all points of $P(k'_s+1, k'_w)$ $(= P(k'_s+1, k_s))$ were finalized. Hence, the fifth invariant holds.

The sixth invariant.. Our goal is to show that for any p_j with $j \in [1, k_s]$, p_j is in I_j.

Note that the position of p_j is not changed for any $j \le k'_s$ when we process the interval I_i. Since the same invariant holds before we process I_i, p_j is in I_j for any $j \in [1, k'_s]$. Hence, if $k_s = k'_s$, we are done with proof. Otherwise, it is sufficient to show that p_j is in I_j for any $j \in [k_{s'}+1, k_s]$.

For $j = k_s$, since $p_j = l_j$, it is trivially true that p_j is in I_j. In the following, we assume $j \in [k'_h, k'_{h+1})$ for some $h \in [s, w-1]$ (recall that $k_s = k'_w$).

According to our algorithm, $p_j = l_{k'_h} + \frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h} \cdot (j - k'_h)$. Let $d'_{\min}$ be the value of $d_{\min}$ before I_i is processed. Let p'_j be the original "temporary" location of p_j before I_i is processed. Since the eighth invariant holds before I_i is processed, we have $p'_j = l_{k'_s} + d'_{min} \cdot (j - k'_s)$ and $p'_j \in I_j$.

We first show that $p_j \le p'_j$, i.e., comparing with its original location, p_j has been moved leftwards in the step of processing I_i. This can be easily seen from the intuitive understanding of the algorithm. We provide a formal proof below.

Since Invariant (8) holds before I_i is processed, $p_{k'_{s+1}}$ was implicitly set to $l_{k'_s} + d'_{\min} \cdot (k'_{s+1} - k'_s)$, which is in $I_{k'_{s+1}}$. Hence, $l_{k'_s} + d'_{\min} \cdot (k'_{s+1} - k'_s) \ge l_{k'_{s+1}}$. Thus, $d'_{\min} \ge \frac{l_{k'_{s+1}} - l_{k'_s}}{k'_{s+1} - k'_s}$. Consequently, $p'_j = l_{k'_s} + d'_{min} \cdot (j - k'_s) \ge l_{k'_s} + \frac{l_{k'_{s+1}} - l_{k'_s}}{k'_{s+1} - k'_s} \cdot (j - k'_s)$.

Since the priority property holds for $\mathscr{L}'$, by Observation 4.2.3, $\frac{k'_{s+1} - k'_s}{l_{k'_{s+1}} - l_{k'_s}} \ge \frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h}$. Hence, $p_j = l_{k'_h} + \frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h} \cdot (j - k'_h) \le l_{k'_h} + \frac{k'_{s+1} - k'_s}{l_{k'_{s+1}} - l_{k'_s}} \cdot (j - k'_h)$.

Now to prove $p_j \le p'_j$, it is sufficient to prove $\frac{k'_{s+1} - k'_s}{l_{k'_{s+1}} - l_{k'_s}} \ge \frac{l_{k'_h} - l_{k'_s}}{k'_h - k'_s}$, which is true by Inequality (4.1) (replacing h and j in Inequality (4.1) by s and k'_h, respectively) due to the priority property of $\mathscr{L}'$.

The above proves that $p_j \leq p'_j$. Since $p'_j \in I_j$, $p'_j \leq r_j$, and thus, $p_j \leq r_j$. To prove $p_j \in I_j$, it remains to prove $p_j \geq l_j$.

If $j = k'_h$, then $p_j = l_j$ and we are done with the proof. Otherwise, due to the priority property of $\mathscr{L}'$ and by applying Inequality (4.1), we have $\frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h} > \frac{l_j - l_{k'_h}}{j - k'_h}$. Therefore, $p_j = l_{k'_h} + \frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h} \cdot (j - k'_h) > l_j$.

This proves that p_j is in I_j. Thus, the sixth invariant holds.

The seventh invariant.. The goal is to show that the distance of any pair of adjacent points of $P(1, k_s)$ is at least $d_{\min}$.

Let $d'_{\min}$ be the value of $d_{\min}$ before we process I_i. We first prove $d'_{\min} > d_{\min}$.

Indeed, if $k_s = k'_s$, then since the eighth invariant holds before I_i is processed, $d'_{\min} = \frac{p'_{i-1} - l_{k_s}}{i-1-k_s}$, where p'_{i-1} is the location of p_{i-1} before we process I_i. Recall that $p'_{i-1} + d'_{\min} > r_i$. Hence, we have $d'_{\min} > \frac{r_i - d'_{\min} - l_{k_s}}{i-1-k_s}$. We can further deduce $d'_{\min} > \frac{r_i - l_{k_s}}{i-k_s}$. Since $d_{\min} = \frac{r_i - l_{k_s}}{i-k_s}$, we obtain $d'_{\min} > d_{\min}$.

If $k_s \neq k'_s$, since k'_{w-1} was removed from $\mathscr{L}$, Inequality (4.8) must hold for $s = w-1$, i.e., $\frac{l_{k'_w} - l_{k'_{w-1}}}{k'_w - k'_{w-1}} > \frac{r_i - l_{k'_{w-1}}}{i - k'_{w-1}}$. Note that for any four positive numbers a, b, c, d with $\frac{a}{b} > \frac{c}{d}$, $a < c$, and $b < d$, it always holds that $\frac{a}{b} > \frac{c-a}{d-b}$. Applying this to the above inequality gives us $\frac{l_{k'_w} - l_{k'_{w-1}}}{k'_w - k'_{w-1}} > \frac{r_i - l_{k'_w}}{i - k'_w}$.

Since $d_{\min} = \frac{r_i - l_{k_s}}{i-k_s}$ and $k_s = k'_w$, we obtain $\frac{l_{k'_w} - l_{k'_{w-1}}}{k'_w - k'_{w-1}} > d_{\min}$.

On the other hand, before I_i is processed, according to the eighth invariant, $l_{k'_s} + d'_{\min} \cdot (k'_{s+1} - k'_s)$ is in $I_{k'_{s+1}}$. Hence, $l_{k'_s} + d'_{\min} \cdot (k'_{s+1} - k'_s) \geq l_{k'_{s+1}}$ and $d'_{\min} \geq \frac{l_{k'_{s+1}} - l_{k'_s}}{k'_{s+1} - k'_s}$.

Further, due to the priority property of $\mathscr{L}'$ and by Observation 4.2.3, it holds that $\frac{k'_{s+1} - k'_s}{l_{k'_{s+1}} - l_{k'_s}} \geq \frac{l_{k'_w} - l_{k'_{w-1}}}{k'_w - k'_{w-1}}$.

Combining our above discussions, we obtain $d'_{\min} > d_{\min}$.

Next, we proceed to prove Invariant (7).

Since Invariant (7) holds before I_i is processed, the distance of every pair of adjacent points of $P(1, k'_s)$ is at least $d'_{\min}$. To prove that the distance of every pair of adjacent points of $P(1, k_s)$ is at least $d_{\min}$, since $d'_{\min} > d_{\min}$, if $k_s = k'_s$, then we are done with the proof, otherwise it is sufficient to show that the distance of every pair of adjacent points of $P(k'_s, k_s)$ is at least $d_{\min}$.

Consider any $h \in [s, w-1]$. When k'_h is removed from $\mathscr{L}$, according to the finalization step, every pair of adjacent points of $P(k'_h, k'_{h+1})$ is $\frac{l_{k'_{h+1}} - l_{k_{h'}}}{k'_{h+1} - k'_h}$. Due to the priority property of $\mathscr{L}'$ and by Observation 4.2.3, $\frac{l_{k'_{h+1}} - l_{k'_h}}{k'_{h+1} - k'_h} \geq \frac{l_{k'_w} - l_{k'_{w-1}}}{k'_w - k'_{w-1}}$. Recall that we have proved above that $\frac{l_{k'_w} - l_{k'_{w-1}}}{k'_w - k'_{w-1}} > d_{\min}$. Hence, we obtain that the distance of every pair of adjacent points of $P(k'_h, k'_{h+1})$ is at least $d_{\min}$. This further implies that the distance of every pair of adjacent points of $P(k'_s, k'_w)$ $(= P(k'_s, k_s))$ is at least $d_{\min}$.

Hence, the seventh invariant holds.

The eighth invariant.. Consider any $j \in [k_s, i]$. Based on our algorithm, p_j is implicitly set to $l_{k_s} + d_{\min} \cdot (j - k_s)$. Hence, to prove the invariant, it remains to show that p_j is in I_j.

If $j = i$, then since $p_i = r_i$, it is true that $p_j \in I_j$. In the following, we assume $j \leq i - 1$.

Let p'_j be the "temporary" location of p_j before I_i is processed. Since the eighth invariant holds before I_i is processed, $p'_j = l_{k'_s} + d'_{\min} \cdot (j - k'_s)$ and $p'_j \in I_j$. Again, let $d'_{\min}$ be the value of $d_{\min}$ before we process I_i. Recall that we have proved above that $d'_{\min} > d_{\min}$.

We claim that $p_j \leq p'_j$. Indeed, if $k_s = k'_s$, then $p_j \leq p'_j$ follows from $d'_{\min} > d_{\min}$. Otherwise, note that $p'_j = l_{k'_s} + d'_{\min} \cdot (k'_w - k'_s) + d'_{\min} \cdot (j - k'_w) = p'_{k'_w} + d'_{\min} \cdot (j - k'_w)$, where $p'_{k'_w}$ is the "temporary" location of $p_{k'_w}$ before I_i is processed. Since $k'_w = k_s$, we have $p'_j = p'_{k_s} + d'_{\min} \cdot (j - k_s)$.

Since Invariant (8) holds before I_i is processed, p'_{k_s} is in I_{k_s}. Hence, $p'_{k_s} \geq l_{k_s}$. Therefore, we obtain $p'_j \geq l_{k_s} + d'_{\min} \cdot (j - k_s) \geq l_{k_s} + d_{\min} \cdot (j - k_s) = p_j$.

This proves the above claim that $p_j \leq p'_j$.

Since $p'_j \in I_j$ and $p_j \leq p'_j$, we obtain $p_j \leq r_j$. To prove $p_j \in I_j$, it remains to show $p_j \geq l_j$, as follows.

According to our algorithm, k_s was not removed from $\mathscr{L}$ either because k_s is the last element of $\mathscr{L}'$ or because Inequality (4.8) is not true.

In the former case, it holds that $k_s = i - 1$. Since $j \in [k_s, i-1]$, $j = k_s$. Due to $p_{k_s} = l_{k_s}$, we obtain $p_j \geq l_j$.

In the latter case, k_s is not the last element of $\mathscr{L}'$ that is in $\mathscr{L}$. Since $k'_w = k_s$, we have $k'_{w+1} = k_{s+1}$. Due to the priority property of $\mathscr{L}'$ and by Inequality (4.1) (with

$h = w$), we have $\frac{l_{k'_{w+1}} - l_{k'_w}}{k'_{w+1} - k'_w} \geq \frac{l_j - l_{k'_w}}{j - k'_w}$. Since $k_s = k'_w$ and $k_{s+1} = k'_{w+1}$, it holds that $\frac{l_{k_{s+1}} - l_{k_s}}{k_{s+1} - k_s} \geq \frac{l_j - l_{k_s}}{j - k_s}$. Since Inequality (4.8) is not true, we further obtain $\frac{r_i - l_{k_s}}{i - k_s} \geq \frac{l_j - l_{k_s}}{j - k_s}$. Recall that $d_{\min} = \frac{r_i - l_{k_s}}{i - k_s}$. Hence, $d_{\min} \geq \frac{l_j - l_{k_s}}{j - k_s}$ and $p_j = l_{k_s} + d_{\min} \cdot (j - k_s) \geq l_j$.

This proves that the eighth invariant holds.

The ninth invariant.. Our goal is to prove that the priority property holds for $\mathscr{L}$. Since the priority property holds for $\mathscr{L}'$, intuitively we only need to take care of the "influence" of i (i.e., some elements were possibly removed from the rear of $\mathscr{L}'$ and i was added to the rear in the rear-processing procedure). Note that although some elements were also possibly removed from the front of $\mathscr{L}'$ in the front-processing procedure, this does not affect the priority property of the remaining elements of the list. Hence, to prove that the priority property holds for $\mathscr{L}$, we have exactly the same situation as in Lemma 4.2.5. Hence, we can use the same proof as that for Lemma 4.2.5. We omit the details.

This proves that all algorithm invariants hold after I_i is processed. The lemma thus follows. □

The above describes a general step of the algorithm for processing the interval I_i. In addition, if $i = n$ and $k_s < n$, we also need to perform the following additional finalization step: for each $j \in [k_s + 1, n]$, we explicitly compute $p_j = l_{k_s} + d_{\min} \cdot (j - k_s)$ and finalize it. This finishes the algorithm.

4.2.4 The Correctness and the Time Analysis

Based on the algorithm invariants and Corollary 4.2.2, the following lemma proves the correctness of the algorithm.

Lemma 4.2.7. *The algorithm correctly computes an optimal solution.*

Proof. Suppose $P = \{p_1, p_2, \ldots, p_n\}$ is the set of points computed by the algorithm. Let $d_{\min}$ be the value and $\mathscr{L} = \{k_s, k_{s+1}, \ldots, k_t\}$ be the critical list after the algorithm finishes.

We first show that for each $j \in [1, n]$, p_j is in I_j. According to the sixth algorithm invariant of $\mathscr{L}$, for each $j \in [1, k_s]$, p_j is in I_j. If $k_s = n$, then we are done with the proof. Otherwise, for each $j \in [k_s + 1, n]$, according to the additional finalization step after I_n is processed, $p_j = l_{k_s} + d_{\min} \cdot (j - k_s)$, which is in I_j by the eighth algorithm invariant.

Next we show that the distance of every pair of adjacent points of P is at least $d_{\min}$. By the seventh algorithm invariant, the distance of every pair of adjacent points of $P(1, k_s)$ is at least $d_{\min}$. If $k_s = n$, then we are done with the proof. Otherwise, it is sufficient to show that the distance of every pair of adjacent points of $P(k_s, n)$ is at least $d_{\min}$, which is true according to the additional finalization step after I_n is processed.

The above proves that P is a *feasible solution* with respect to $d_{\min}$, i.e., all points of P are in their corresponding intervals and the distance of every pair of adjacent points of P is at least $d_{\min}$.

To show that P is also an optimal solution, based on the second algorithm invariant, it holds that $d_{\min} = \frac{r_{j^*} - l_{i^*}}{j^* - i^*}$. By Corollary 4.2.2, $d_{\min}$ is an optimal objective value. Therefore, P is an optimal solution. □

The running time of the algorithm is analyzed in the proof of Theorem 4.2.8. The pseudocode is given in Algorithm 1.

Theorem 4.2.8. *Our algorithm computes an optimal solution of the line version of points dispersion problem in $O(n)$ time.*

Proof. In light of Lemma 4.2.7, we only need to show that the running time of the algorithm is $O(n)$.

To process an interval I_i, according to our algorithm, we only spend $O(1)$ time in addition to two possible procedures: a front-processing procedure and a rear-processing procedure. Note that the front-processing procedure may contain several finalization steps. There may also be an additional finalization step after I_n is processed. For the purpose of analyzing the total running time of the algorithm, we exclude the finalization steps from the front-processing procedures.

For processing I_i, the front-processing procedure (excluding the time of the finalization steps) runs in $O(k+1)$ time where k is the number of elements removed from the front of the critical list $\mathscr{L}$. An easy observation is that any element can be removed from $\mathscr{L}$ at most once in the entire algorithm. Hence, the total time of all front-processing procedures in the entire algorithm is $O(n)$.

Algorithm 1: The algorithm for the line version of the problem

Input: n intervals $I_1, I_2, \ldots, I_n$ sorted from left to right on ℓ

Output: n points $p_1, p_2, \ldots, p_n$ with $p_i \in I_i$ for each $1 \leq i \leq n$

```
p_1 ← l_1, i* ← 1, j* ← 1, d_min ← ∞, 𝓛 ← {1};
for i ← 2 to n do
    if p_{i-1} + d_min ≤ l_i then
        p_i ← l_i, 𝓛 ← {i};
    else
        if l_i < p_{i-1} + d_min ≤ r_i then
            p_i ← p_{i-1} + d_min;
        else /* p_{i-1} + d_min > r_i */
            p_i ← r_i, k_s ← the front element of 𝓛;
            while |𝓛| > 1 do            /* the front-processing procedure */
                if (l_{k_{s+1}} - l_{k_s})/(k_{s+1} - k_s) > (r_i - l_{k_s})/(i - k_s) then
                    for j ← k_s + 1 to k_{s+1} do
                        p_j ← l_{k_s} + (l_{k_{s+1}} - l_{k_s})/(k_{s+1} - k_s) · (j - k_s);
                    remove k_s from 𝓛,   k_s ← the front element of 𝓛;
                else
                    break;
            i* ← k_s, j* ← i, d_min ← (r_{j*} - l_{i*})/(j* - i*);
        while |𝓛| > 1 do                /* the rear-processing procedure */
            k_t ← the rear element of 𝓛;
            if (l_{k_t} - l_{k_{t-1}})/(k_t - k_{t-1}) > (l_i - l_{k_{t-1}})/(i - k_{t-1}) then break ;
            remove k_t from 𝓛;
        add i to the rear of 𝓛;
k_s ← the front element of 𝓛;
if k_s < n then
    for j ← k_s + 1 to n do
        p_j ← l_{k_s} + d_min · (j - k_s);
```

Similarly, for processing I_i, the rear-processing procedure runs in $O(k + 1)$ time where k is the number of elements removed from the rear of $\mathscr{L}$. Again, since any element can be removed from $\mathscr{L}$ at most once in the entire algorithm, the total time of all rear-processing procedures in the entire algorithm is $O(n)$.

Clearly, each point is finalized exactly once in the entire algorithm. Hence, all finalization steps in the entire algorithm together take $O(n)$ time.

Therefore, the algorithm runs in $O(n)$ time in total. □

4.3 The Cycle Version

In the cycle version, the intervals of $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ in their index order are sorted cyclically on $\mathcal{C}$. Recall that the intervals of $\mathcal{I}$ are pairwise disjoint.

For each $i \in [1, n]$, let l_i and r_i denote the two endpoints of I_i, respectively, such that if we move from l_i to r_i clockwise on $\mathcal{C}$, we will always stay on I_i.

For any two points p and q on $\mathcal{C}$, we use $|\overrightarrow{pq}|$ to denote the length of the arc of $\mathcal{C}$ from p to q clockwise, and thus the distance of p and q on $\mathcal{C}$ is $\min\{|\overrightarrow{pq}|, |\overrightarrow{qp}|\}$.

For each interval $I_i \in \mathcal{I}$, we use $|I_i|$ to denote its length; note that $|I_i| = |\overrightarrow{l_i r_i}|$. We use $|\mathcal{C}|$ to denote the total length of $\mathcal{C}$.

Our goal is to find a point p_i in I_i for each $i \in [1, n]$ such that the minimum distance between any pair of these points, i.e., $\min_{1 \leq i < j \leq n} |p_i p_j|$, is maximized.

Let $P = \{p_1, p_2, \ldots, p_n\}$ and let d_{opt} be the optimal objective value. It is obvious that $d_{opt} \leq \frac{|\mathcal{C}|}{n}$. Again, for simplicity of discussion, we make a general position assumption that no two endpoints of the intervals have the same location on $\mathcal{C}$.

4.3.1 The Algorithm

The main idea is to convert the problem to a problem instance on a line and then apply our line version algorithm. More specifically, we copy all intervals of $\mathcal{I}$ twice to a line ℓ and then apply our line version algorithm on these $2n$ intervals. The line version algorithm will find $2n$ points in these intervals. We will show that a subset of n points in n consecutive intervals correspond to an optimal solution for our original problem on $\mathcal{C}$. The details are given below.

Let ℓ be the x-axis. For each $1 \leq i \leq n$, we create an interval $I'_i = [l'_i, r'_i]$ on ℓ with $l'_i = |\overrightarrow{l_1 l_i}|$ and $r'_i = l'_i + |I_i|$, which is actually a copy of I_i. In other words, we first put a copy I'_1 of I_1 at ℓ such that its left endpoint is at 0 and then we continuously copy other intervals to ℓ in such a way that the pairwise distances of the intervals on ℓ are the same as the corresponding clockwise distances of the intervals of $\mathcal{I}$ on $\mathcal{C}$. The above only makes one copy for each interval of $\mathcal{I}$. Next, we make another copy for each interval of $\mathcal{I}$ in a similar way: for each $1 \leq i \leq n$, we create an interval $I'_{i+n} = [l'_{i+n}, r'_{i+n}]$ on ℓ with $l'_{i+n} = l'_i + |\mathcal{C}|$ and $r'_{i+n} = r'_i + |\mathcal{C}|$. Let $\mathcal{I}' = \{I'_1, I'_2, \ldots, I'_{2n}\}$. Note that the intervals of $\mathcal{I}'$ in their index order are sorted from left to right on ℓ.

We apply our line version algorithm on the intervals of $\mathcal{I}'$. However, a subtle change is that here we initially set $d_{\min} = \frac{|\mathcal{C}|}{n}$ instead of $d_{\min} = \infty$. The rest of the algorithm is the same as before. We want to emphasize that this change on initializing $d_{\min}$ is necessary to guarantee the correctness of our algorithm for the cycle version. A consequence of this change is that after the algorithm finishes, if $d_{\min}$ is still equal to $\frac{|\mathcal{C}|}{n}$, then $\frac{|\mathcal{C}|}{n}$ may not be the optimal objective value for the above line version problem, but if $d_{\min} < \frac{|\mathcal{C}|}{n}$, then $d_{\min}$ must be the optimal objective value. As will be clear later, this does not affect our final solution for our original problem on the cycle $\mathcal{C}$. Let $P' = \{p'_1, \ldots, p'_{2n}\}$ be the points computed by the line version algorithm with $p'_i \in I'_i$ for each $i \in [1, 2n]$.

Let k be the largest index in $[1, n]$ such that $p'_k = l'_k$. Note that such an index k always exists since $p'_1 = l'_1$. Due to that we initialize $d_{\min} = \frac{|\mathcal{C}|}{n}$ in our line version algorithm, we can prove the following lemma.

Lemma 4.3.1. *It holds that* $p'_{k+n} = l'_{k+n}$.

Proof. We prove the lemma by contradiction. Assume to the contrary that $p'_{k+n} \neq l'_{k+n}$. Since $p'_{k+n} \in I'_{k+n}$, it must be that $p'_{k+n} > l'_{k+n}$. Let p'_i be the rightmost point of P' to the left of p'_{k+n} such that p'_i is at the left endpoint of its interval I'_i. Depending on whether $i \leq n$, there are two cases.

1. If $i > n$, then let $j = i - n$. Since $i < k + n$, $j < k$. We claim that $|p'_j p'_k| < |p'_{j+n} p'_{n+k}|$.

   Indeed, since $p'_j \geq l'_j$ and $p'_k = l'_k$, we have $|p'_j p'_k| \leq |l'_j l'_k|$. Note that $|l'_j l'_k| = |l'_{j+n} l'_{k+n}|$. On the other hand, since $p'_{j+n} = l'_{j+n}$ and $p'_{k+n} > l'_{k+n}$, it holds that $|p'_{j+n} p'_{k+n}| > |l'_{j+n} l'_{k+n}|$. Therefore, the claim follows.

   Let d be the value of $d_{\min}$ right before the algorithm processes I'_i. Since during the execution of our line version algorithm $d_{\min}$ is monotonically decreasing, it holds that $|p'_j p'_k| \geq d \cdot (k - j)$. Further, by the definition of i, for any $m \in [i+1, k+n]$, $p'_m > l'_m$. Thus, according to our line version algorithm, the distance of every adjacent pair of points of $p'_i, p'_{i+1} \ldots, p'_{k+n}$ is at most d. Thus, $|p'_i p'_{k+n}| \leq d \cdot (k+n-i)$. Since $j = i - n$, we have $|p'_{j+n} p'_{k+n}| \leq d \cdot (k - j)$. Hence, we obtain $|p'_j p'_k| \geq |p'_{j+n} p'_{k+n}|$. However, this contradicts with our above claim.

2. If $i \leq n$, then by the definition of k, we have $i = k$. Let d be the value of $d_{\min}$ right before the algorithm processes I'_i. By the definition of i, the distance of every adjacent pair of points of $p'_k, p'_{k+1} \ldots, p'_{k+n}$ is at most d. Hence, $|p'_k p'_{k+n}| \leq n \cdot d$. Since $p'_k = l'_k$ and $p'_{n+k} > l'_{n+k}$, we have $|p'_k p'_{n+k}| > |l'_k l'_{n+k}| = |\mathcal{C}|$. Therefore, we obtain that $n \cdot d > |\mathcal{C}|$.

   However, since we initially set $d_{\min} = |\mathcal{C}|/n$ and the value $d_{\min}$ is monotonically decreasing during the execution of the algorithm, it must hold that $n \cdot d \leq |\mathcal{C}|$. We thus obtain contradiction.

Therefore, it must hold that $p'_{n+k} = l'_{n+k}$. The lemma thus follows. □

We construct a solution set P for our cycle version problem by mapping the points $p'_k, p'_{k+1}, \ldots, p'_{n+k-1}$ back to $\mathcal{C}$. Specifically, for each $i \in [k, n]$, we put p_i at a point on $\mathcal{C}$ with a distance $p'_i - l'_i$ clockwise from l_i; for each $i \in [1, k-1]$, we put p_i at a point on $\mathcal{C}$ at a distance $p'_{i+n} - l'_{i+n}$ clockwise from l_i. Clearly, p_i is in I_i for each $i \in [1, n]$. Hence, P is a "feasible" solution for our cycle version problem. Below we show that P is actually an optimal solution.

Consider the value $d_{\min}$ returned by the line version algorithm after all intervals of $\mathcal{I}'$ are processed. Since the distance of every pair of adjacent points of $p'_k, p'_{k+1}, \ldots, p'_{n+k}$ is at least $d_{\min}$, $p'_k = l'_k$, $p'_{n+k} = l'_{n+k}$ (by Lemma 4.3.1), and $|l'_k l'_{n+k}| = |\mathcal{C}|$, by our way of constructing P, the distance of every pair of adjacent points of P on $\mathcal{C}$ is at least $d_{\min}$.

Recall that d_{opt} is the optimal object value of our cycle version problem. The following lemma implies that P is an optimal solution.

Lemma 4.3.2. $d_{\min} = d_{opt}$.

Proof. Since P is a feasible solution with respect to $d_{\min}$, $d_{\min} \leq d_{opt}$ holds.

If $d_{\min} = |\mathcal{C}|/n$, since $d_{opt} \leq |\mathcal{C}|/n$, we obtain $d_{opt} \leq d_{\min}$. Therefore, $d_{opt} = d_{\min}$, which leads to the lemma.

In the following, we assume $d_{\min} \neq |\mathcal{C}|/n$. Hence, $d_{\min} < |\mathcal{C}|/n$. According to our line version algorithm, there must exist $i^* < j^*$ such that $d_{\min} = \frac{r'_{j^*} - l'_{i^*}}{j^* - i^*}$. We assume there is no i with $i^* < i < j^*$ such that $d_{\min} = \frac{r'_{j^*} - l'_i}{j^* - i}$ since otherwise we could change

i^* to i. Since $d_{\min} = \frac{r'_{j^*} - l'_{i^*}}{j^* - i^*}$, it is necessary that $p'_{i^*} = l'_{i^*}$ and $p'_{j^*} = r'_{j^*}$. By the above assumption, there is no $i \in [i^*, j^*]$ such that $p'_i = l'_i$. Since $p'_k = l'_k$ and $p'_{k+n} = l'_{k+n}$ (by Lemma 4.3.1), one of the following three cases must be true: $j^* < k$, $k \leq i^* < j^* < n+k$, or $n + k \leq i^*$. In any case, $j^* - i^* < n$. By our way of defining r'_{j^*} and l'_{i^*}, we have the following:

$$d_{\min} = \frac{r'_{j^*} - l'_{i^*}}{j^* - i^*} = \begin{cases} |\overrightarrow{l_{i^*} r_{j^*}}|/(j^* - i^*), & \text{if } j^* \leq n, \\ |\overrightarrow{l_{i^*} r_{j^*-n}}|/(j^* - i^*), & \text{if } i^* \leq n < j^*, \\ |\overrightarrow{l_{i^*-n} r_{j^*-n}}|/(j^* - i^*) & \text{if } n < i^*. \end{cases}$$

We claim that $d_{opt} \leq d_{\min}$ in all three cases: $j^* \leq n$, $i^* \leq n < j^*$, and $n < i^*$. In the following we only prove the claim in the first case where $j^* \leq n$ since the other two cases can be proved analogously (e.g., by re-numbering the indices).

Our goal is to prove $d_{opt} \leq \frac{|\overrightarrow{l_{i^*} r_{j^*}}|}{j^* - i^*}$. Consider any optimal solution in which the solution set is $P = \{p_1, p_2, \ldots, p_n\}$. Consider the points $p_{i^*}, p_{i^*+1}, \ldots, p_{j^*}$, which are in the intervals $I_{i^*}, I_{i^*+1}, \ldots, I_{j^*}$. Clearly, $|\overrightarrow{p_k p_{k+1}}| \geq d_{opt}$ for any $k \in [i^*, j^*-1]$. Therefore, we have $|\overrightarrow{p_{i^*} p_{j^*}}| \geq d_{opt} \cdot (j^* - i^*)$. Note that $|\overrightarrow{p_{i^*} p_{j^*}}| \leq |\overrightarrow{l_{i^*} r_{j^*}}|$. Consequently, we obtain $d_{opt} \leq \frac{|\overrightarrow{l_{i^*} r_{j^*}}|}{j^* - i^*}$.

Since both $d_{\min} \leq d_{opt}$ and $d_{opt} \leq d_{\min}$, it holds that $d_{opt} = d_{\min}$. The lemma thus follows. □

The above shows that P is an optimal solution with $d_{opt} = d_{\min}$. The running time of the algorithm is $O(n)$ because the line version algorithm runs in $O(n)$ time. As a summary, we have the following theorem.

Theorem 4.3.3. *The cycle version of the points dispersion problem is solvable in $O(n)$ time.*

4.4 Concluding Remarks

In this chapter we present a linear time algorithm for the point dispersion problem on disjoint intervals on a line. Further, by making use of this algorithm, we also solve the same problem on a cycle in linear time.

It would be interesting to consider the general case of the problem in which the intervals may overlap. In fact, for the line version, if we know the order of the intervals in which the sought points in an optimal solution are sorted from left to right, then we can apply our algorithm to process the intervals in that order and the obtained solution is an optimal solution. For example, if no interval is allowed to contain another completely, then there must exist an optimal solution in which the sought points from left to right correspond to the intervals ordered by their left (or right) endpoints. Hence, to solve the general case of the line version problem, the key is to find an order of intervals. This is also the case for the cycle version.

CHAPTER 5

MULTIPLE BARRIER COVERAGE

5.1 Introduction

In this chapter, we study algorithms for the problems for covering multiple barriers. These are basic geometric problems and have applications in barrier coverage of mobile sensors in wireless sensor networks. For convenience, in the following we introduce and discuss the problems from the mobile sensor barrier coverage point of view. The results in this chapter have been published in a conference [22].

5.1.1 Problem Definitions and Our Results

Let L be a line, say, the x-axis. Let $\mathcal{B}$ be a set of m pairwise disjoint segments, called *barriers*, sorted on L from left to right. Let S be a set of n sensors in the plane, and each sensor $s_i \in S$ is represented by a point (x_i, y_i). If a sensor is moved on L, it has a *sensing/covering range* of length r, i.e., if a sensor s is located at x on L, then all points of L in the interval $[x - r, x + r]$ are *covered* by s and the interval is called the *covering interval* of s. The problem is to move all sensors of S onto L such that each point of every barrier is covered by at least one sensor and the maximum movement of all sensors of S is minimized, i.e., the value $\max_{s_i \in S} \sqrt{(x_i - x_i')^2 + y_i^2}$ is minimized, where x_i' is the location of s_i on L in the solution (its y-coordinate is 0 since L is the x-axis). We call it the *multiple-barrier coverage* problem, denoted by MBC.

We assume that covering range of the sensors is long enough so that a coverage of all barriers is always possible. Note that we can check whether a coverage is possible in $O(m + n)$ time by an easy greedy algorithm (e.g., try to cover all barriers one by one from left to right using sensors in such a way that their covering intervals do not overlap except at their endpoints).

Previously, only the special case $m = 1$ was studied and the problem was solved in

$O(n^3 \log n)$ time [57]. In this chapter, we propose an $O(n^2 \log n \log\log n + nm \log m)$-time algorithm for any m, which improves the algorithm in [57] by almost a linear factor even for the special case $m = 1$.

We further consider a *line-constrained* version of the problem where all sensors of S are initially on L. Previously, only the special case $m = 1$ was studied and the problem was solved in $O(n \log n)$ time [33]. We present an $O((n+m)\log(n+m))$ time algorithm for any m, and the running time matches that of the algorithm in [33] when $m = 1$.

5.1.2 Related Work

Sensors are basic units in wireless sensor networks. The advantage of allowing the sensors to be mobile increases monitoring capability compared to those static ones. One of the most important applications in mobile wireless sensor networks is to monitor a barrier to detect intruders in an attempt to cross a specific region. Barrier coverage [57, 58], which guarantees that every movement crossing a barrier of sensors will be detected, is known to be an appropriate model of coverage for such applications. Mobile sensors normally have limited battery power and therefore their movements should be as small as possible.

Dobrev et al. [59] studies several problems on covering multiple barriers in the plane. They showed that these problems are generally NP-hard when sensors have different ranges. They also proposed polygonal-time algorithms for several special cases of the problems, e.g., barriers are parallel or perpendicular to each other, and sensors have some constrained movements. In fact, if sensors have different ranges, by an easy reduction from the Partition Problem as in [59], we can show that our problem MBC is NP-hard even for the line-constrained version and $m = 2$.

Other previous work has been focused on the line-constrained problem with $m = 1$. Czyzowicz et al. [60] first gave an $O(n^2)$ time algorithm, and later, Chen et al. [33] solved the problem in $O(n \log n)$ time. If sensors have different ranges, Chen et al. [33] presented an $O(n^2 \log n)$ time algorithm. For the *weighted case* where sensors have weights such that the moving cost of a sensor is its moving distance times its weight, Wang and Zhang [61] gave an $O(n^2 \log n \log\log n)$ time algorithm for the case where sensors have the same range.

The *min-sum* version of the line-constrained problem with $m = 1$ has also been studied, where the objective is to minimize the sum of the moving distances of all sensors. If sensors have different ranges, then the problem is NP-hard [62]. Otherwise, Czyzowicz et al. [62] gave an $O(n^2)$ time algorithm, and Andrews and Wang [63] improved the algorithm to $O(n \log n)$ time. The *min-num* version of the problem was also studied, where the goal is to move the minimum number of sensors to form a barrier coverage. Mehrandish et al. [16, 17] proved that the problem is NP-hard if sensors have different ranges and gave polynomial time algorithms otherwise.

Bhattacharya et al. [13] studied a circular barrier coverage problem in which the barrier is a circle and the sensors are initially located inside the circle. The goal is to move sensors to the circle to form a regular n-gon (so as to cover the circle) such that the maximum sensor movement is minimized. An $O(n^{3.5} \log n)$-time algorithm was given in [13] and later Chen et al. [14] improved the algorithm to $O(n \log^3 n)$ time. The min-sum version of the problem was also studied [13, 14].

5.1.3 Our Approach

To solve the problem MBC, one major difficulty is that we do not know the order of the sensors of S on L in an optimal solution. Therefore, our main effort is to find such an order. To this end, we first develop a *decision algorithm* that can determine whether $\lambda \geq \lambda^*$ for any value λ, where λ^* is the maximum sensor movement in an optimal solution. Our decision algorithm runs in $O(m + n \log n)$ time. Then, we solve the problem MBC by "parameterizing" the decision algorithm in a way similar in spirit to parametric search [64]. The high-level scheme of our algorithm is very similar to those in [33, 61], but many low-level computations are different.

The line-constrained version of the problem is much easier due to an *order preserving property*: there exists an optimal solution in which the order of the sensors is the same as in the input. This leads to a linear-time decision algorithm using the greedy strategy. Also based on this property, we can find a set Λ of $O(n^2 m)$ "candidate values" such that Λ contains λ^*. To avoid computing Λ explicitly, we implicitly organize the elements of Λ into $O(n)$ sorted arrays such that each array element can be found in $O(\log m)$ time. Finally, by applying the matrix search technique in [65], along with our linear-time decision algorithm, we compute λ^* in $O((n+m)\log(n+m))$ time. We should point out that implicitly organizing the elements of Λ into sorted arrays is the key and also the major difficulty for solving the problem, and our technique may be interesting in its own right.

The rest of this chapter is organized as follows. We introduce some notation in Section 6.3. In Section 5.3, we present our algorithm for the line-constrained problem. In Section 5.4, we present our decision algorithm for the problem MBC. Section 5.5 solves the problem MBC. We conclude this chapter in Section 5.6, with remarks that our techniques can be used to reduce the space complexities of some previous algorithms in [33, 61].

5.2 Preliminaries

We denote the barriers of $\mathcal{B}$ by $B_1, B_2, \ldots, B_m$ sorted on L from left to right. For each B_i, let a_i and b_i denote the left and right endpoints of B_i, respectively. For ease of exposition, we make a general position assumption that $a_i \neq b_i$ for each B_i. The degenerated case can also be handled by our techniques, but the discussions would be more tedious.

With a little abuse of notation, for any point x on L (the x-axis), we also use x to denote its x-coordinate, and vice versa. We assume that the left endpoint of B_1 is at 0, i.e., $a_1 = 0$. Let β denote the right endpoint of B_m, i.e., $\beta = b_m$.

We denote the sensors of S by $s_1, s_2, \ldots, s_n$ sorted by their x-coordinates. For each sensor s_i located on a point x of L, $x - r$ and $x + r$ are the left and right endpoints of the covering interval of s_i, respectively, and we call them the *left and right extensions* of s_i, respectively.

Again, let λ^* be the maximum sensor movement in an optimal solution. Given λ, the *decision problem* is to determine whether $\lambda \geq \lambda^*$, or equivalently, whether we can move each sensor with distance at most λ such that all barriers can be covered. If yes, we say that λ is a *feasible value.* Thus, we also call it a *feasibility test* on λ.

5.3 The Line-Constrained Version of MBC

In this section, we present our algorithm for the line-constrained MBC. As in the special case $m = 1$ [60], a useful observation is that the *order preserving* property holds: There exists an optimal solution in which the order of the sensors is the same as in the input. Due to this property, we first give a linear-time greedy algorithm for feasibility tests.

Lemma 5.3.1. *Given any $\lambda > 0$, we can determine whether λ is a feasible value in $O(n+m)$ time.*

Proof. We first move every sensor rightwards for distance λ. Then, every sensor is allowed to move leftwards at most 2λ but is not allowed to move rightwards any more. Next we use a greedy strategy to move sensors leftwards as small as possible to cover the currently uncovered leftmost barrier. To this end, we maintain a point p on a barrier that we need to cover such that all barrier points to the left of p are covered but the barrier points to the right of p are not. We consider the sensors s_i and the barriers B_j from left to right.

Initially, $i = j = 1$ and $p = a_1$. In general, suppose p is located at a barrier B_j and we are currently considering s_i. If p is at β, then we are done and λ is feasible. If p is located at b_j and $j \neq m$, then we move p rightwards to a_{j+1} and proceed with $j = j + 1$. In the following, we assume that p is not at b_j. Let $x_i^r = x_i + \lambda$, i.e., the location of s_i after it is moved rightwards by λ.

1. If $x_i^r + r \leq p$, then we proceed with $i = i + 1$.
2. If $x_i^r - r \leq p < x_i^r + r$, we move p rightwards to $x_i^r + r$.
3. If $x_i^r - 2\lambda - r \leq p < x_i^r - r$, then we move s_i leftwards such that the left extension of s_i is at p, and we then move p to the right extension of s_i.
4. If $p < x_i^r - 2\lambda - r$, then we stop the algorithm and report that λ is not feasible.

Suppose the above moved p rightwards (i.e., in the second and third cases). Then, if $p \geq \beta$, we report that λ is feasible. Otherwise, if p is not on a barrier, then we move p rightwards to the left endpoint of the next barrier. In either case, p is now located at a barrier, denoted by B_j, and we increase i by one. We proceed as above with B_j and s_i. It is easy to see that the algorithm runs in $O(n+m)$ time. □

Let OPT be an optimal solution that preserves the order of the sensors. For each $i \in [1, n]$, let x'_i be the position of s_i in OPT. We say that a set of k sensors are in *attached positions* if the union of their covering intervals is a single interval of length equal to $2rk$. The following lemma is self-evident and is an extension of a similar observation for the case $m = 1$ in [60].

Lemma 5.3.2. *There exists a sequence of sensors $s_i, s_{i+1}, \ldots, s_j$ in attached positions in OPT such that one of the following three cases holds. (a) The sensor s_j is moved to the left by distance λ^* and $x'_i = a_k + r$ for some barrier B_k (i.e., the sensors from s_i to s_j together cover the interval $[a_k, a_k + 2r(j - i + 1)]$). (b) The sensor s_i is moved to the right by λ^* and $x'_j = b_k - r$ for some barrier B_k. (c) The sensor s_i is moved rightwards by λ^* and s_j is moved leftwards by λ^*.*

Cases (a) and (b) are symmetric in the above lemma. Let Λ_1 be the set of all possible distance values introduced by s_j in Case (a). Specifically, for any pair (i, j) with $1 \leq i \leq j \leq n$ and any barrier B_k with $1 \leq k \leq m$, define $\lambda(i, j, k) = x_j - (a_k + 2r(j-i) + r)$. Let Λ_1 consists of $\lambda(i, j, k)$ for all such triples (i, j, k). We define Λ_2 symmetrically be the set of all possible values introduced by s_i in Case (b). We define Λ_3 as the set consisting of the values $[x_j - x_i - 2r(j - i)]/2$ for all pairs (i, j) with $1 \leq i < j \leq n$. Clearly, $|\Lambda_3| = O(n^2)$ and both $|\Lambda_1|$ and $|\Lambda_2|$ are $O(mn^2)$. Let $\Lambda = \Lambda_1 \cup \Lambda_2 \cup \Lambda_3$.

By Lemma 5.3.2, λ^* is in Λ, and more specifically, λ^* is the smallest feasible value of Λ. Hence, we can first compute Λ and then find the smallest feasible value in Λ by using the decision algorithm. However, that would take $\Omega(mn^2)$ time. To reduce the time, we will not compute Λ explicitly, but implicitly organize the elements of Λ into certain sorted arrays and then apply the matrix search technique proposed in [65], which has been widely used, e.g., [66, 67]. Since we only need to deal with sorted arrays instead of more general matrices, we review the technique with respect to arrays in the following lemma.

Lemma 5.3.3. [65] *Given a set of N sorted arrays of size at most M each, we can compute the smallest feasible value of these arrays with $O(\log N + \log M)$ feasibility tests and the total time of the algorithm excluding the feasibility tests is $O(\tau \cdot N \cdot \log \frac{2M}{N})$, where τ is* *the time for evaluating each array element (i.e., the number of array elements that need to be evaluated is $O(N \cdot \log \frac{2M}{N})$).*

With Lemma 5.3.3, we can compute the smallest feasible values in the three sets Λ_1, Λ_2, and Λ_3, respectively, and then return the smallest one as λ^*. For Λ_3, Chen et al. [33] (see Lemma 14) gave an approach to order in $O(n \log n)$ time the elements of Λ_3 into $O(n)$ sorted arrays of $O(n)$ elements each such that each array element can be obtained in $O(1)$ time. Consequently, by applying Lemma 5.3.3, the smallest feasible value of Λ_3 can be computed in $O((n+m)\log n)$ time.

For Λ_1 and Λ_2, in the case $m = 1$, the elements of each set can be easily ordered into $O(n)$ sorted arrays of $O(n)$ elements each [33]. However, in our problem for general m, the problem becomes significantly more difficult if we want to obtain a subquadratic-time algorithm. Indeed, this is the main challenge of our method. In what follows, our main effort is to prove the following lemma.

Lemma 5.3.4. *For the set Λ_1, in $O(m \log m)$ time, we can implicitly form a set $\mathcal{A}$ of $O(n)$ sorted arrays of $O(m^2 n)$ elements each such that each array element can be computed in $O(\log m)$ time and every element of Λ_1 is contained in one of the arrays. The same applies to the set Λ_2.*

We note that our technique for Lemma 5.3.4 might be interesting in its own right and may find other applications as well. Before proving Lemma 5.3.4, we first prove the following result..

Theorem 5.3.5. *The line-constrained version of MBC can be solved in $O((n+m)\log(n+m))$ time.*

Proof. It is sufficient to compute λ^*, after which we can apply the decision algorithm on λ^* to obtain an optimal solution.

Let Λ'_1 denote the set of all elements in the arrays of $\mathcal{A}$ specified in Lemma 5.3.4. Define Λ'_2 similarly with respect to Λ_2. By Lemma 5.3.4, $\Lambda_1 \subseteq \Lambda'_1$ and $\Lambda_2 \subseteq \Lambda'_2$. Since $\lambda^* \in \Lambda_1 \cup \Lambda_2 \cup \Lambda_3$, we also have $\lambda^* \in \Lambda'_1 \cup \Lambda'_2 \cup \Lambda_3$. Hence, λ^* is the smallest feasible value in $\Lambda'_1 \cup \Lambda'_2 \cup \Lambda_3$. Let λ_1, λ_2, and λ_3 be the smallest feasible values in the sets Λ'_1, Λ'_2, and Λ_3, respectively. As discussed before, λ_3 can be computed in $O((n+m)\log n)$ time. By

Lemma 5.3.4, applying the algorithm in Lemma 5.3.3 can compute both λ_1 and λ_2 in $O((n+m)(\log m+\log n))$ time. Note that $(n+m)(\log m+\log n) = \Theta((n+m)\log(n+m))$. The theorem thus follows. □

5.3.1 Proving Lemma 5.3.4

In this section, we prove Lemma 5.3.4. We will only prove the case for Λ_1, since the other case for Λ_2 is symmetric. Recall that $\Lambda_1 = \{\lambda(i,j,k) \mid 1 \leq i \leq j \leq n, 1 \leq k \leq m\}$, where $\lambda(i,j,k) = x_j - (a_k + 2r(j-i) + r)$.

For any j and k, let $A[j,k]$ denote the list $\lambda(i,j,k)$ for $i = 1, 2, \ldots, j$, which is sorted in increasing order. With a little abuse of notation, let $A[j]$ denote the union of the elements in $A[j,k]$ for all $k \in [1,m]$. Clearly, Λ_1 is the union of $A[j]$ for all $1 \leq j \leq n$. In the following, we will organize the elements in each $A[j]$ into a sorted array $B[j]$ of size $O(nm^2)$ such that given any index t, the t-th element of $B[j]$ can be computed in $O(\log m)$ time, which will prove Lemma 5.3.4. Our technique replies on the following property: the difference of every two adjacent elements in each list $A[j,k]$ is the same, i.e., $2r$.

Notice that for any $k \in [1, m-1]$, the first element of $A[j,k]$ is larger than the first element of $A[j,k+1]$, and similarly, the last element of $A[j,k]$ is larger than the last element of $A[j,k+1]$. Hence, the first element of $A[j,m]$, i.e., $\lambda(1,j,m)$, is the smallest element of $A[j]$ and the last element of $A[j,1]$, i.e., $\lambda(j,j,1)$, is the largest element of $A[j]$. Let $\lambda_{min}[j] = \lambda(1,j,m)$ and $\lambda_{max}[j] = \lambda(j,j,1)$.

For each $k \in [1,m]$, we extend the list $A[j,k]$ to a new sorted list $B[j,k]$ with the following property: (1) $A[j,k]$ is a sublist of $B[j,k]$; (2) the difference every two adjacent elements of $B[j,k]$ is $2r$; (3) the first element of $B[j,k]$ is in $[\lambda_{min}[j], \lambda_{min}[j]+2r)$; (4) the last element of $B[j,k]$ is in $(\lambda_{max}[j]-2r, \lambda_{max}[j]]$. Specifically, $B[j,k]$ is defined as follows. Note that $\lambda(1,j,k)$ and $\lambda(j,j,k)$ are the first and last elements of $A[j,k]$, respectively. We let $\lambda(1,j,k) - \lfloor\frac{\lambda(1,j,k)-\lambda_{min}[j]}{2r}\rfloor \cdot 2r$ and $\lambda(j,j,k) + \lfloor\frac{\lambda_{max}[j]-\lambda(j,j,k)}{2r}\rfloor \cdot 2r$ be the first and last elements of $B[j,k]$, respectively. Then, the h-th element of $B[j,k]$ is equal to $\lambda(1,j,k) - \lfloor\frac{\lambda(1,j,k)-\lambda_{min}[j]}{2r}\rfloor \cdot 2r + 2r \cdot (h-1)$ for any $h \in [1, \alpha[j]]$, where $\alpha[j] = 1 + \lceil\frac{\lambda_{\max}[j]-\lambda_{min}[j]}{2r}\rceil$. Hence, $B[j,k]$ has $\alpha[j]$ elements. One can verify that $B[j,k]$ has the above four properties. Note that we can implicitly create the lists $B[j,k]$ in

$O(1)$ time so that given any $k \in [1, m]$ and $h \in [1, \alpha[j]]$, we can obtain the h-th element of $B[j, k]$ in $O(1)$ time. Let $B[j]$ be the sorted list of all elements of $B[j, k]$ for all $1 \le k \le m$. Hence, $B[j]$ has $\alpha[j] \cdot m$ elements.

Let σ_j be the permutation of $1, 2, \ldots, m$ following the sorted order of the first elements of $B[j, k]$. For any $k \in [1, m]$, let $\sigma_j(k)$ be the k-th index in σ_j. We have the following lemma.

Lemma 5.3.6. *For any t with $1 \le t \le \alpha[j] \cdot m$, the t-th smallest element of $B[j]$ is the h_t-th element of the list $B[j, \sigma_j(k_t)]$, where $h_t = \lceil \frac{t}{m} \rceil$ and $k_t = t \mod m$.*

Proof. Consider any h with $1 \le h \le \alpha[j]$. Denote by $B_h[j, k]$ the h-th element of $B[j, k]$ for each $k \in [1, m]$. By our definition of $B[j, k]$, $B_h[j, k] \in [\lambda_{\min}[j] + 2r(h-1), \lambda_{\min}[j] + 2rh)$. Therefore, for any $h' < h$, it holds that $B_{h'}[j, k] < B_h[j, k']$ for any k and k' in $[1, m]$. On the other hand, by the definition of σ_j, $B_h[j, \sigma(k)] < B_h[j, \sigma(k')]$ for any $1 \le k < k' \le m$.

Based on the above discussion, one can verify that the lemma statement holds. □

By the preceding lemma, if the permutation σ_j is known, we can obtain the t-th smallest element of $B[j]$ in $O(1)$ time for any index t. Computing σ_j can be done in $O(m \log m)$ time by sorting. If we apply the sorting algorithm on every $j \in [1, n]$, then we wound need $O(nm \log m)$ time. Fortunately, the following lemma implies that we only need to do the sorting once.

Lemma 5.3.7. *The permutation σ_j is unique for all $j \in [1, n]$.*

Proof. Consider any j_1, j_2 in $[1, n]$ with $j_1 \ne j_2$ and any k_1, k_2 in $[1, m]$ with $k_1 \ne k_2$. For any j and k, let $B_1[j, k]$ denote the first element of $B[j, k]$. To prove the lemma, it is sufficient to show that $B_1[j_1, k_1] < B_1[j_1, k_2]$ if and only if $B_1[j_2, k_1] < B_1[j_2, k_2]$.

Recall that $B_1[j, k] = \lambda(1, j, k) - \lfloor \frac{\lambda(1,j,k) - \lambda_{min}[j]}{2r} \rfloor \cdot 2r$ and $\lambda(1, j, k) = x_j - (a_k + 2rj - r)$. Thus, $B_1[j, k] = x_j - a_k + r - \lfloor \frac{x_j - a_k + r - \lambda_{min}[j]}{2r} \rfloor \cdot 2r$. Further, since $\lambda_{min}[j] = \lambda(1, j, m) = x_j - (a_m + 2rj - r)$, $B_1[j, k] = x_j - a_k + r - \lfloor \frac{a_m - a_k + 2rj}{2r} \rfloor \cdot 2r = x_j - a_k + r - \lfloor \frac{a_m - a_k}{2r} \rfloor \cdot 2r - 2rj$.

Therefore, $B_1[j_1, k_1] - B_1[j_1, k_2] = a_{k_2} - a_{k_1} + (\lfloor \frac{a_m - a_{k_2}}{2r} \rfloor - \lfloor \frac{a_m - a_{k_1}}{2r} \rfloor) \cdot 2r$ and $B_1[j_2, k_1] - B_1[j_2, k_2] = a_{k_2} - a_{k_1} + (\lfloor \frac{a_m - a_{k_2}}{2r} \rfloor - \lfloor \frac{a_m - a_{k_1}}{2r} \rfloor) \cdot 2r$. Hence, $B_1[j_1, k_1] -$

$B_1[j_1, k_2] = B_1[j_2, k_1] - B_1[j_2, k_2]$, which implies that $B_1[j_1, k_1] < B_1[j_1, k_2]$ if and only if $B_1[j_2, k_1] < B_1[j_2, k_2]$. □

In summary, after $O(m \log m)$ time preprocessing to compute the permutation σ_j for any j, we can form the arrays $B[j]$ for all $j \in [1, n]$ such that given any $j \in [1, n]$ and $t \in [1, \alpha[j] \cdot m]$, we can compute t-th smallest element of $B[j]$ in $O(1)$ time. However, we are not done yet, because we do not have a reasonable upper bound for $\alpha[j]$, which is equal to $1 + \lceil \frac{\lambda_{\max}[j] - \lambda_{min}[j]}{2r} \rceil = 1 + \lceil \frac{\lambda(j,j,1) - \lambda(1,j,m)}{2r} \rceil = j + \lceil \frac{a_m - a_1}{2r} \rceil$. To address the issue, in the sequel, we will partition the indices $k \in [1, m]$ into groups and then apply our above approach to each group so that the corresponding $\alpha[j]$ values can be bounded, e.g., by $O(mn)$.

The Group Partition Technique.. We consider any index $j \in [1, m]$.

We partition the indices $1, 2, \ldots, m$ into groups each consisting of a sequence of consecutive indices, such that each group has the following *intra-group overlapping property*: For any index k that is not the largest index in the group, the first element of $A[j, k]$ is smaller than or equal to the last element of $A[j, k+1]$, i.e., $\lambda(1, j, k) \leq \lambda(j, j, k+1)$. Further, the groups have the following *inter-group non-overlapping property*: For the largest index k in a group that is not the last group, the first element of $A[j, k]$ is larger than the last element of $A[j, k+1]$, i.e., $\lambda(1, j, k) > \lambda(j, j, k+1)$.

We compute the groups in $O(m)$ time as follows. Initially, add 1 into the first group G_1. Let $k = 1$. While the first element of $A[j, k]$ is smaller than or equal to the last element of $A[j, k+1]$, we add $k+1$ into G_1 and reset $k = k+1$. After the while loop, G_1 is computed. Then, starting from $k+1$, we compute G_2 and so on until index m is included in the last group. Let $G_1, G_2, \ldots, G_l$ be the l groups we have computed. Note that $l \leq m$.

Consider any group G_g with $1 \leq g \leq l$. We process the lists $A[j][k]$ for all $k \in G_g$ in the same way as discussed before. Specifically, for each $k \in G_g$, we create a new list $B[j][k]$ from $A[j][k]$. Based on the new lists in the group G_g, we form the sorted array $B_g[j]$ with a total of $|G_g| \cdot \alpha_g[j]$ elements, where $|G_g|$ is the number of indices of G_g and $\alpha_g[j]$ is corresponding $\alpha[j]$ value as defined before but only on the group G_g, i.e., if k_1 and k_2 are the smallest and largest indices of G_g respectively, then $\alpha_g[j] =$

$1 + \lceil \frac{\lambda(j,j,k_1)-\lambda(1,j,k_2)}{2r} \rceil$. Let $B[j]$ be the sorted list of all elements in the lists $B_g[j]$ for all groups. Due to the intra-group overlapping property of each group, it holds that $\alpha_g \le |G_g| \cdot n$. Thus, the size of $B[j]$ is at most $\sum_{g=1}^{l} |G_g|^2 \cdot n$, which is at most $m^2 n$ since $\sum_{g=1}^{l} |G_g| = m$.

Suppose we want to find the t-th smallest element of $B[j]$. As preprocessing, we compute a sequence of values $\beta_g[j]$ for $g = 1, 2, \ldots, l$, where $\beta_g[j] = \sum_{g'=1}^{g} \alpha_{g'}[j] \cdot |G_{g'}|$, in $O(m)$ time. To compute the t-th smallest element of $B[j]$, we first do binary search on the sequence $\beta_1[j], \beta_2[j], \ldots, \beta_l[j]$ to find in $O(\log l)$ time the index g such that $t \in (\beta_{g-1}[j], \beta_g[j]]$. Due to the inter-group non-overlapping property of the groups, the t-th smallest element of $B[j]$ is the $(t - \beta_{g-1}[j])$-th element in the array $B_g[j]$, which can be found in $O(1)$ time. As $l \le m$, the total time for computing the t-th smallest element of $B[j]$ is $O(\log m)$.

The above discussion is on any single index $j \in [1, n]$. With $O(m \log m)$ time preprocessing, given any t, we can find the t-th smallest value of $B[j]$ in $O(\log m)$ time.

For all indices $j \in [1, n]$, it appears that we have to do the group partition for every $j \in [1, n]$, which would take quadratic time. To resolve the problem, we show that it is sufficient to only use the group partition based on $j = n$ for all other $j \in [1, n-1]$. The details are given below.

Suppose from now on $G_1, G_2, \ldots, G_l$ are the groups computed as above with respect to $j = n$. We know that the inter-group non-overlapping property holds respect to the index n. The following lemma shows that the property also holds with respect to any other index $j \in [1, n-1]$.

Lemma 5.3.8. *The inter-group non-overlapping property holds for any* $j \in [1, n-1]$.

Proof. Consider any $j \in [1, n-1]$ and any k that is the largest index in a group G_g with $g \in [1, l-1]$. The goal is to show that the first element of $A[j, k]$ is larger than the last element of $A[j, k+1]$, i.e., $\lambda(1, j, k) > \lambda(j, j, k+1)$. Since the groups are defined with respect to the index n, it holds that $\lambda(1, n, k) > \lambda(n, n, k+1)$.

Recall that $\lambda(i, j, k) = x_j - (a_k + 2r(j-i) + r)$. Therefore, $\lambda(1, j, k) - \lambda(j, j, k+1) = a_{k+1} - a_k + 2r(1-j)$ and $\lambda(1, n, k) - \lambda(n, n, k+1) = a_{k+1} - a_k + 2r(1-n)$. Since

$\lambda(1,n,k) > \lambda(n,n,k+1)$, $a_{k+1} - a_k + 2r(1-n) > 0$. As $j < n$, $a_{k+1} - a_k + 2r(1-j) > 0$, and thus $\lambda(1,j,k) > \lambda(j,j,k+1)$. □

Consider any group G_g with $1 \leq g \leq l$ and any $j \in [1,n]$. For each $k \in G_g$, we create a new list $B[j][k]$ based on $A[j][k]$ in the same way as discussed before. Based on the new lists, we form the sorted array $B_g[j]$ of $|G_g| \cdot \alpha_g[j]$ elements. We also define the value $\beta_g[j]$ in the same way as before. The following lemma shows that $\alpha_g[j]$ and $\beta_g[j]$ can be computed based on $\alpha_g[n]$ and $\beta_g[n]$.

Lemma 5.3.9. *For any $j \in [1,n-1]$ and $g \in [1,l]$, $\alpha_g[j] = \alpha_g[n] - n + j$ and $\beta_g[j] = \beta_g[n] + \delta_g \cdot g \cdot (j-n)$, where $\delta_g = \sum_{g'=1}^{g} |G_{g'}|$.*

Proof. Consider any $g \in [1,l]$. Let k_1 and k_2 be the smallest and the largest indices in G_g, respectively. By definition, $\alpha_g[j] = 1 + \lceil \frac{\lambda(j,j,k_1) - \lambda(1,j,k_2)}{2r} \rceil = 1 + \lceil \frac{a_{k_2} - a_{k_1} + 2r(j-1)}{2r} \rceil = j + \lceil \frac{a_{k_2} - a_{k_1}}{2r} \rceil$. Therefore, for any $j \in [1,n-1]$, $\alpha_g[j] = \alpha_g[n] - n + j$.

By definition, $\beta_g[j] = \alpha_1[j] \cdot |G_1| + \alpha_2[j] \cdot |G_2| + \cdots + \alpha_g[j] \cdot |G_g| = (\alpha_1[n] - n + j) \cdot |G_1| + (\alpha_2[n] - n + j) \cdot |G_2| + \cdots + (\alpha_g[n] - n + j) \cdot |G_g| = \beta_g[n] + (j-n) \cdot g \cdot (|G_1| + |G_2| + \cdots + |G_g|) = \beta_g[n] + \delta_g \cdot g \cdot (j-n)$. □

For each group G_g, we compute the permutation for the lists $B[n,k]$ for all k in the group. Computing the permutations for all groups takes $O(m \log m)$ time. Also as preprocessing, we first compute δ_g, $\alpha_g(n)$ and $\beta_g(n)$ for all $g \in [1,l]$ in $O(m)$ time. By Lemma 5.3.9, for any $j \in [1,n]$ and any $g \in [1,l]$, we can compute $\alpha_g[j]$ and $\beta_g[j]$ in $O(1)$ time. Because the lists $B[n,k]$ for all k in each group G_g have the intra-group overlapping property, it holds that $\alpha_g[n] \leq |G_g| \cdot n$. Hence, $\sum_{g=1}^{l} \alpha_g[n] \leq mn$. For any $j \in [1,n-1]$, by Lemma 5.3.9, $\alpha_g[j] < \alpha_g[n]$, and thus $\sum_{g=1}^{l} \alpha_g[j] \leq mn$. Recall that $B[j]$ is the sorted array of all elements in $B_g[j]$ for $g \in [1,l]$. Thus, $B[j]$ has at most $m^2 n$ elements.

For any $j \in [1,n]$ and any $t \in [1, \sum_{g=1}^{l} |G_g| \cdot \alpha_g[j]]$, suppose we want to compute the t-th smallest element of $B[j]$. Due to the inter-group non-overlapping property in Lemma 5.3.8, we can still use the previous binary search approach. For the running time, since we can obtain each $\beta_g[j]$ for any $g \in [1,l]$ in $O(1)$ time by Lemma 5.3.9, we can still compute the t-th smallest element of $B[j]$ in $O(\log m)$ time.

This proves Lemma 5.3.4.

5.4 The Decision Problem of MBC

In this section, we present an $O(m+n \log n)$-time algorithm for the decision problem of MBC: given any value $\lambda > 0$, determine whether $\lambda \geq \lambda^*$. Our algorithm for MBC in Section 5.5 will make use of this decision algorithm. The decision problem may have independent interest because in some applications each sensor has a limited energy λ and we want to know whether their energy is enough for them to move to cover all barriers.

Consider any value $\lambda > 0$. We assume that $\lambda \geq \max_{1 \leq i \leq n} |y_i|$ since otherwise some sensor cannot reach L by moving λ (and thus λ is not feasible). For any sensor $s_i \in S$, define $x_i^r = x_i + \sqrt{\lambda^2 - y_i^2}$ and $x_i^l = x_i - \sqrt{\lambda^2 - y_i^2}$. Note that x_i^r and x_i^l are respectively the rightmost and leftmost points of L s_i can reach with respect to λ. We call x_i^r the *rightmost (resp., leftmost) λ-reachable location* of s_i on L. For any point x on L, we use $p^+(x)$ to denote a point x' such that $x' > x$ and x' is infinitesimally close to x.

The high-level scheme of our algorithm is similar to that in [61]. We first describe the algorithm and then show its correctness. Finally, we discuss its implementation.

5.4.1 The Algorithm Description

We use a *configuration* to refer to a specification on where each sensor $s_i \in S$ is located. For example, in the *input configuration*, each s_i is at (x_i, y_i).

We begin with moving each sensor s_i to x_i^r on L. Let C_0 denote the resulting configuration. In C_0, each sensor s_i is not allowed to move rightwards but can move leftwards on L by a maximum distance $2\sqrt{\lambda^2 - y_i^2}$.

If $\lambda \geq \lambda^*$, our algorithm will compute a subset of sensors with their new locations to cover all barriers of $\mathcal{B}$ and the maximum movement of each sensor of in the subset is at most λ.

For each step i with $i \geq 1$, let C_{i-1} be the configuration right before the i-th step. Our algorithm maintains the following *invariants*. (1) We have a subset of sensors $S_{i-1} = \{s_{g(1)}, s_{g(2)}, \ldots, s_{g(i-1)}\}$, where for each $1 \leq j \leq i-1$, $g(j)$ is the index of the sensor $s_{g(j)}$ in S. (2) In C_{i-1}, each sensor s_k of S_{i-1} is at a new location $x'_k \in [x_k^l, x_k^r]$, and all other sensors are still in their locations of C_0. (3) A value R_{i-1} is maintained such that $0 \leq R_{i-1} < \beta$, R_{i-1} is on a barrier, every barrier point $x < R_{i-1}$ is covered by a sensor of S_{i-1} in C_{i-1}. (4) If R_{i-1} is not at the left endpoint of a barrier, then R_{i-1} is covered by a sensor of S_{i-1} in C_{i-1}. (5) The point $p^+(R_{i-1})$ is not covered by any sensor in S_{i-1}.

Initially when $i = 1$, we let $S_0 = \emptyset$ and $R_0 = 0$, and thus all algorithm invariants hold for C_0. The i-th step of the algorithm finds a sensor $s_{g(i)} \in S \setminus S_{i-1}$ and moves it to a new location $x'_{g(i)} \in [x^l_{g(i)}, x^r_{g(i)}]$ and thus obtains a new configuration C_i. The details are given below.

Define S_{i1} to be the set of sensors that cover the point $p^+(R_{i-1})$ in C_{i-1}, i.e., $S_{i1} = \{s_k \mid x^r_k - r \leq R_{i-1} < x^r_k + r\}$. By the algorithm invariant (5), no sensor in S_{i-1} covers $p^+(R_{i-1})$. Thus, $S_{i1} \subseteq S \setminus S_{i-1}$. If $S_{i1} \neq \emptyset$, then we choose an *arbitrary* sensor in S_{i1} as $s_{g(i)}$ (e.g., see Fig. 5.1) and let $x'_{g(i)} = x^r_{g(i)}$. We then set $R_i = x'_{g(i)} + r$, i.e., R_i is at the right endpoint of the covering interval of $s_{g(i)}$. Note that C_i is C_{i-1} because $s_{g(i)}$ is not moved.

If $S_{i1} = \emptyset$, then we define $S_{i2} = \{s_k \mid x^l_k - r \leq R_{i-1} < x^r_k - r\}$ (i.e., S_{i2} consists of those sensors s_k that does not cover R_{i-1} when it is at x^r_k but is possible to do so when it is at some location in $[x^l_k, x^r_k]$). If $S_{i2} \neq \emptyset$, we choose the *leftmost* sensor of S_{i2} as $s_{g(i)}$ (e.g., see Fig. 5.2), and let $x'_{g(i)} = R_{i-1} + r$ (i.e., we move $s_{g(i)}$ to $x'_{g(i)}$ and thus obtain C_i). If $S_{i2} = \emptyset$, then we conclude that $\lambda < \lambda^*$ and terminate the algorithm.

Hence, if $S_{i1} = S_{i2} = \emptyset$, the algorithm will stop and report $\lambda < \lambda^*$. Otherwise, a sensor $s_{g(i)}$ is found from either S_{i1} or S_{i2}, and it is moved to $x'_{g(i)}$. In either case, $R_i = x'_{g(i)} + r$ and $S_i = S_{i-1} \cup \{s_{g(i)}\}$. If $R_i \geq \beta$, then we terminate the algorithm and report $\lambda \geq \lambda^*$. Otherwise, we further perform the following *jump-over procedure*: We check whether R_i is located at the interior of any barrier; if not, then we set R_i to the left endpoint of the barrier right after R_i.

This finishes the i-th step of our algorithm. One can verify that all algorithm invariants are maintained. As there are n sensors in S, the algorithm will finish in at most n steps.

5.4.2 The Algorithm Correctness

The correctness proof is similar to that for the algorithm in [61], so we briefly discuss it.

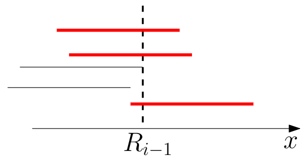


Figure 5.1. Illustrating the set S_{i1}. The covering intervals of sensors are shown with segments (the red thick segments correspond to the sensors in S_{i1}). Every sensor in S_{i1} can be $s_{g(i)}$.

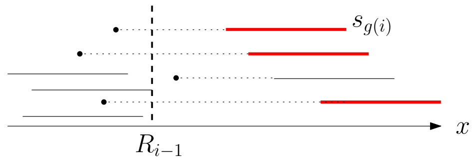


Figure 5.2. Illustrating the set S_{i2}. The segments are the covering intervals of sensors. The red thick segments correspond to the sensors in S_{i2}. The four black points corresponding to the values $x_k^l - r$ of the four sensors x_k to the right of R_{i-1}. The sensor $s_{g(i)}$ is labeled.

If the decision algorithm reports $\lambda \geq \lambda^*$, say, in the i-th step, then according to our algorithm, the configuration C_i is a feasible solution. Below, we show that if the algorithm reports $\lambda < \lambda^*$, then λ is indeed not a feasible value.

We first note that due to our jump-over procedure and our general position assumption, R_i cannot be at the right endpoint of a barrier, and thus $p^+(R_i)$ must be a point of a barrier.

An interval on L is said to be *left-aligned* if its left side is closed and equal to 0 and its right side is open. The algorithm correctness will be easily shown with the following Lemma 5.4.1. The proof of the lemma is very similar to Lemma 1 in [61], so we omit it.

Lemma 5.4.1. *Consider any configuration C_i. Suppose S'_i is the set of sensors in S whose right extensions are at most R_i in C_i. Then, the interval $[0, R_i)$ is the largest possible left-aligned interval such that all barrier points in the interval can be covered by the sensors of S'_i with respect to λ (i.e., the moving distance of each sensor of S'_i is at most λ).*

Suppose our algorithm reports $\lambda < \lambda^*$ in the i-th step. We show that λ is not a feasible value. Indeed, according to our algorithm, $R_{i-1} < \beta$ and $S_{i1} = S_{i2} = \emptyset$ in the configuration C_{i-1}. Let S'_{i-1} be the set of sensors whose right extensions are at most R_{i-1} in C_{i-1}. On the one hand, by Lemma 5.4.1 (replacing index i in the lemma by $i-1$), $[0, R_{i-1})$ is the largest left-aligned interval such that all barrier points in the interval that can be covered by the sensors in S'_{i-1}. On the other hand, since both S_{i1} and S_{i2} are empty, no sensor in $S \setminus S'_{i-1}$ can cover the point $p^+(R_{i-1})$. Recall that $p^+(R_{i-1})$ is a barrier point not covered by any sensor in S_{i-1}. Due to $R_{i-1} < \beta$, we conclude that sensors of S cannot cover all barrier points in the interval $[0, p^+(R_{i-1})] \subseteq [0, \beta]$ with respect to λ. Thus, λ is not a feasible value. This establishes the correctness of our decision algorithm.

5.4.3 The Algorithm Implementation

The implementation is similar to that in [61] and we briefly discuss it. We first implement the algorithm in $O(m+n\log n)$ time, and then we reduce the time to $O(m+n\log\log n)$ under certain assumption.

We first move each sensor s_i to x_i^r and thus obtain the configuration C_0. Then, we sort the extensions of all sensors in C_0 together with the endpoints of all barriers. To maintain the set S_{i1} during the algorithm, we sweep a point p on L from left to right. During the sweeping, when p encounters the left (resp., right) extension of a sensor, we insert the sensor into S_{i1} (resp., delete it from S_{i1}). In this way, in each i-th step of the algorithm, when p is at R_{i-1}, S_{i1} is available.

If $S_{i1} \neq \emptyset$, we pick an arbitrary sensor in S_{i1} as $s_{g(i)}$. To store the set S_{i1}, since sensors have the same range, the earlier a sensor is inserted into S_{i1}, the earlier it is deleted from S_{i1}. Thus, we can simply use a first-in-first-out queue to store S_{i1} such that each insertion/deletion can be done in constant time. We can always pick the front sensor in the queue as $s_{g(i)}$.

If $S_{i1} = \emptyset$, then we need to compute S_{i2}. To maintain S_{i2} during the sweeping of p, we do the following. Initially when we do the sorting as discussed above, we also sort the n values $x_i^l - r$ for all $1 \leq i \leq n$. During the sweeping of p, if p encounters a point $x_k^l - r$ for some sensor s_k, we insert s_k to S_{i2}, and if p encounters a left extension of some sensor s_k, we delete s_k from S_{i2}. In this way, when p is at R_{i-1}, S_{i2} is available. If $S_{i2} \neq \emptyset$, we need to find the leftmost sensor in S_{i2} as $s_{g(i)}$, for which we use a balanced binary search tree T to store all sensors of S_{i2} where the "key" of each sensor s_k is the value x_k^r. T can support each of the following operations on S_{i2} in $O(\log n)$ time: inserting a sensor, deleting a sensor, finding the leftmost sensor.

If $s_{g(i)}$ is from S_{i1}, then we do not need to move $s_{g(i)}$. We proceed to sweep p as usual. If $s_{g(i)}$ is from S_{i2}, we need to move $s_{g(i)}$ leftwards to $x'_{g(i)} = R_{i-1} + r$. Since $s_{g(i)}$ is moved, we should also update the original sorted list including the extensions of all sensors in C_0 to guide the future sweeping of p. To avoid the explicit update, we use a flag table for all sensor extensions in C_0. Initially, every table entry is *valid.* If $s_{g(i)}$ is moved, then we set the table entries of the two extensions of the sensor *invalid.* During the sweeping of p, when p encounters a sensor extension, we first check the table to see whether the extension is still valid. If yes, then we proceed as usual; otherwise we ignore the event. This only costs extra constant time for each event. In addition, we calculate R_i as discussed before, and the jump-over procedure can be implemented in $O(1)$ time since the barrier endpoints are also sorted.

To analyze the running time, since the barriers are given sorted on L, the sorting step takes $O(m+n\log n)$. Since there are $O(n)$ operations on the tree T, the total time of the algorithm is $O(m+n\log n)$. Thus we obtain the following result.

Theorem 5.4.2. *Given any value λ, we can determine whether $\lambda \geq \lambda^*$ in $O(m+n\log n)$ time.*

Our algorithm in Section 5.5 will perform feasibility tests multiple times, for which we have the following result.

Lemma 5.4.3. *Suppose the values x_i^r for all $i = 1, 2, \ldots, n$ are already sorted, we can determine whether $\lambda \geq \lambda^*$ in $O(m + n\log\log n)$ time for any λ.*

Proof. Our $O(m + n\log n)$ time implementation is dominated by two parts. The first part is the sorting. The second part is on performing the operations on the set S_{i2}, each taking $O(\log n)$ time by using the tree T. The rest of the algorithm together takes $O(n + m)$ time. Now that the values x_i^r for all $i = 1, 2, \ldots, n$ are already sorted, the sorting step takes $O(n + m)$ time since the barriers are already given sorted.

Recall that the keys of the sensors of T are the values x_k^r. Let $Q = \{x_k^r \mid 1 \leq k \leq n\}$. For each sensor s_k, we use $rank(s_k)$ to denote the *rank* of x_k^r in Q (i.e., $rank(s_k) = t$ if x_k^r is the t-th smallest value in Q). Since Q is already sorted, all sensor ranks can be computed in $O(n)$ time. It is easy to see that the leftmost sensor of T is the sensor with the smallest rank. Therefore, we can also use the ranks as the keys of sensors of T, and the advantage of doing so is that the rank of each sensor is an integer in $[1, n]$. Hence, instead of using a balanced binary search tree, we can use an integer data structure, e.g., the van Emde Boas Tree (or vEB tree for short) [29], to maintain S_{i2}. The vEB tree can support each of the following operations on S_{i2} in $O(\log\log n)$ time [29]: inserting a sensor, deleting a sensor, and finding the sensor with the smallest rank. Using a vEB tree, all operations on S_{i2} in the algorithm can be performed in $O(n \log\log n)$ time. The lemma thus follows. □

5.5 Solving the Problem MBC

In this section, we solve the problem MBC. It suffices to compute λ^*. The high-level scheme of our algorithm is similar to that in [61], although some low-level details are different.

In this section, we use $x_i^r(\lambda)$ to refer to x_i^r for any λ, so that we consider $x_i^r(\lambda)$ as a function on $\lambda \in [0, \infty]$, which actually defines a half of the upper branch (on the right side of the y-axis) of a hyperbola. Let σ be the order of the values $x_i^r(\lambda^*)$ for all $i \in [1, n]$. To make use of Lemma 5.4.3, we first run a preprocessing step in Lemma 5.5.1.

Lemma 5.5.1. *With $O(n \log^3 n + m \log^2 n)$ time preprocessing, we can compute σ and an interval $(\lambda_1^*, \lambda_2^*]$ containing λ^* such that σ is also the order of the values $x_i^r(\lambda)$ for any $\lambda \in (\lambda_1^*, \lambda_2^*]$.*

Proof. To compute σ, we apply Megiddo's parametric search [64] to sort the values $x_i^r(\lambda^*)$ for $i \in [1, n]$, using the decision algorithm in Theorem 5.4.2. Indeed, recall that $x_i^r(\lambda) = x_i + \sqrt{\lambda^2 - y_i^2}$. Hence, as λ increases, $x_i^r(\lambda)$ is a (strictly) increasing function. For any two indices i and j, there is at most one root on $\lambda \in [0, \infty)$ for the equation: $x_i^r(\lambda) = x_j^r(\lambda)$. Therefore, we can apply Megiddo's parametric search [64] to do the sorting. The total time is $O((\tau + n) \log^2 n)$, where τ is the running time of the decision algorithm. By Theorem 5.4.2, $\tau = O(m + n \log n)$. Hence, the total time for computing σ is $O(m \log^2 n + n \log^3 n)$.

In addition, Megiddo's parametric search [64] will return an interval $(\lambda_1^*, \lambda_2^*]$ such that it contains λ^* and σ is also the order of the values $x_i^r(\lambda)$ for any $\lambda \in (\lambda_1^*, \lambda_2^*]$. □

Note that λ^* is the smallest feasible value. As $\lambda^* \in (\lambda_1^*, \lambda_2^*]$, our subsequent feasible tests will be only on values $\lambda \in (\lambda_1^*, \lambda_2^*)$ because if $\lambda \leq \lambda_1^*$, then λ is not feasible and if $\lambda \geq \lambda_2^*$, then λ is feasible. Lemmas 5.4.3 and 5.5.1 together lead to the following result.

Lemma 5.5.2. *Each feasibility test can be done in $O(m + n \log\log n)$ time for any $\lambda \in (\lambda_1^*, \lambda_2^*)$.*

To compute λ^*, we "parameterize" our decision algorithm with λ as a parameter. Although we do not know λ^*, we execute the decision algorithm in such a way that it computes the same subset of sensors $s_{g(1)}, s_{g(2)}, \ldots$ as would be obtained if we ran the decision algorithm on $\lambda = \lambda^*$.

Recall that for any λ, step i of our decision algorithm computes the sensor $s_{g(i)}$, the set $S_i = \{s_{g(1)}, s_{g(2)}, \ldots, s_{g(i)}\}$, and the value R_i, and obtains the configuration C_i. In the following, we often consider λ as a variable rather than a fixed value. Thus, we will use $S_i(\lambda)$ (resp., $R_i(\lambda)$, $s_{g(i)}(\lambda)$, $C_i(\lambda)$, $x_i^r(\lambda)$) to refer to the corresponding S_i (resp., R_i, $s_{g(i)}$, C_i, x_i^r). Our algorithm has at most n steps. Consider a general i-th step for $i \geq 1$. Right before the step, we have an interval $(\lambda_{i-1}^1, \lambda_{i-1}^2]$ and a sensor set $S_{i-1}(\lambda)$, such that the following algorithm invariants hold.

1. $\lambda^* \in (\lambda_{i-1}^1, \lambda_{i-1}^2]$.
2. The set $S_{i-1}(\lambda)$ is the same (with the same order) for all values $\lambda \in (\lambda_{i-1}^1, \lambda_{i-1}^2)$.
3. $R_{i-1}(\lambda)$ on $\lambda \in (\lambda_{i-1}^1, \lambda_{i-1}^2)$ is either constant or equal to $x_j + \sqrt{\lambda^2 - y_j^2} + c$ for some constant c and some sensor s_j with $1 \leq j \leq i-1$, and $R_{i-1}(\lambda)$ is maintained by the algorithm.
4. $R_{i-1}(\lambda) < \beta$ for any $\lambda \in (\lambda_{i-1}^1, \lambda_{i-1}^2)$.

Initially when $i = 1$, we let $\lambda_0^1 = \lambda_1^*$ and $\lambda_0^2 = \lambda_2^*$. Since $S_0(\lambda) = \emptyset$ and $R_0(\lambda) = 0$ for any λ, by Lemma 5.5.1, all invariants hold for $i = 1$. In general, the i-th step will either compute λ^*, or obtain an interval $(\lambda_i^1, \lambda_i^2] \subseteq (\lambda_{i-1}^1, \lambda_{i-1}^2]$ and a sensor $s_{g(i)}(\lambda)$ with $S_i(\lambda) = S_{i-1}(\lambda) \cup \{s_{g(i)}(\lambda)\}$. The running time of the step is $O((m + n\log\log n)(\log n + \log m))$. The details are given below.

5.5.1 The Algorithm

We assume $\lambda^* \neq \lambda_{i-1}^2$ and thus λ^* is in $(\lambda_{i-1}^1, \lambda_{i-1}^2)$. Our following algorithm can proceed without this assumption and we make the assumption only for explaining the rationale of our approach. Since $\lambda^* \in (\lambda_{i-1}^1, \lambda_{i-1}^2)$, according to our algorithm invariants, for all $\lambda \in (\lambda_{i-1}^1, \lambda_{i-1}^2)$, $S_{i-1}(\lambda)$ is the same as $S_{i-1}(\lambda^*)$. We simulate the decision algorithm on $\lambda = \lambda^*$. To determine the sensor $s_{g(i)}(\lambda^*)$, we first compute the set $S_{i1}(\lambda^*)$, as follows.

Consider any sensor s_k in $S \setminus S_{i-1}(\lambda)$. Its position in $C_{i-1}(\lambda)$ is $x_k^r(\lambda) = x_k + \sqrt{\lambda^2 - y_k^2}$, which is an increasing function of λ. Thus, both the left and the right extensions of s_k in $C_{i-1}(\lambda)$ are increasing functions of λ. Suppose $f(\lambda)$ is either the left or the right extension of s_k in $C_{i-1}(\lambda)$. According to our algorithm invariants, $R_{i-1}(\lambda)$ on $\lambda \in (\lambda_{i-1}^1, \lambda_{i-1}^2)$ is either constant or equal to $x_j + \sqrt{\lambda^2 - y_j^2} + c$ for some constant c and some sensor s_j. We claim that there is at most one value λ in $(\lambda_{i-1}^1, \lambda_{i-1}^2)$ such that $R_{i-1}(\lambda) = f(\lambda)$. Indeed, if $R_{i-1}(\lambda)$ is constant, then this is obviously true; otherwise, this is also true because each of $f(\lambda)$ and $R_{i-1}(\lambda)$ on $\lambda \in [0, \infty)$ defines a half branch of a hyperbola (and thus they have at most one intersection in $(\lambda_{i-1}^1, \lambda_{i-1}^2)$).

Let $S' = S \setminus S_{i-1}(\lambda)$. If we increase λ from λ_{i-1}^1 to λ_{i-1}^2, an "event" happens if $R_{i-1}(\lambda)$ is equal to the left or right extension value of a sensor $s_k \in S'$ at some value of λ (called an *event value*), and $S_{i1}(\lambda)$ does not change between any two adjacent events. To compute $S_{i1}(\lambda^*)$, we first compute all event values, and this can be done in $O(n)$ time by using the function $R_{i-1}(\lambda)$ and all left and right extension functions of the sensors in S'. Let Λ denote the set of all event values, and we also add λ_{i-1}^1 and λ_{i-1}^2 to Λ. We then sort all values in Λ. Using the feasibility test in Lemma 5.5.2, we do binary search to find two adjacent values λ_1 and λ_2 in the sorted list of Λ such that $\lambda^* \in (\lambda_1, \lambda_2]$. Note that $(\lambda_1, \lambda_2] \subseteq (\lambda_{i-1}^1, \lambda_{i-1}^2]$. Since $|\Lambda| = O(n)$, the binary search uses $O(\log n)$ feasibility tests, which takes overall $O(m \log n + n \log n \log \log n)$ time.

We make another assumption that $\lambda^* \neq \lambda_2$. Again, this assumption is only for the explanation and the following algorithm can proceed without this assumption. Under the assumption, for any $\lambda \in (\lambda_1, \lambda_2)$, the set $S_{i1}(\lambda)$ is exactly $S_{i1}(\lambda^*)$. Hence, we can compute $S_{i1}(\lambda^*)$ by taking any $\lambda \in (\lambda_1, \lambda_2)$ and explicitly computing $S_{i1}(\lambda)$ in $O(n)$ time.

The above has computed $S_{i1}(\lambda^*)$. If $S_{i1}(\lambda^*) \neq \emptyset$, we take any sensor of $S_{i1}(\lambda^*)$ as $s_{g(i)}(\lambda^*)$. Further, we let $\lambda_i^1 = \lambda_1$, $\lambda_i^2 = \lambda_2$, and $S_i(\lambda) = S_{i-1}(\lambda) \cup \{s_{g(i)}(\lambda^*)\}$.

If $S_{i1}(\lambda^*) = \emptyset$, then we need to compute the set $S_{i2}(\lambda^*)$. Since $\lambda^* \in (\lambda_1, \lambda_2) \subseteq (\lambda_{i-1}^1, \lambda_{i-1}^2)$, according to our algorithm invariants, $R_{i-1}(\lambda)$ is a nondecreasing function on $\lambda \in (\lambda_1, \lambda_2)$. For each sensor $s_k \in S$, $x_k - \sqrt{\lambda^2 - y_k^2} - r$ is a decreasing function on $\lambda \in (\lambda_1, \lambda_2)$. Therefore, the interval (λ_1, λ_2) contains at most one value λ such that $R_{i-1}(\lambda) = x_k - \sqrt{\lambda^2 - y_k^2} - r$. If we increase λ from λ_1 to λ_2, an "event" happens when $R_{i-1}(\lambda)$ is equal to $x_k - \sqrt{\lambda^2 - y_k^2} - r$ for some sensor $s_k \in S'$ at some *event value* λ, and the set $S_{i2}(\lambda)$ is fixed between any two adjacent events. Hence, we use the following way to compute $S_{i2}(\lambda^*)$.

We first compute the set Λ of all event values, and also add λ_1 and λ_2 to Λ. After sorting all values of Λ, by using our decision algorithm, we do binary search to find two adjacent values λ'_1 and λ'_2 in the sorted list of Λ with $\lambda^* \in (\lambda'_1, \lambda'_2]$. Note that $(\lambda'_1, \lambda'_2] \subseteq (\lambda_1, \lambda_2]$. Since $|\Lambda| = O(n)$, the binary search calls the decision algorithm $O(\log n)$ times, which takes $O(m \log n + n \log n \log \log n)$ time in total. Since $S_{i2}(\lambda)$ is the same for all $\lambda \in (\lambda'_1, \lambda'_2)$. We take an arbitrary value $\lambda \in (\lambda'_1, \lambda'_2)$ and compute $S_{i2}(\lambda)$ explicitly in $O(n)$ time.

Lemma 5.5.3. *If $S_{i2}(\lambda) = \emptyset$, then λ^* is in $\{\lambda^2_{i-1}, \lambda_2, \lambda'_2\}$.*

Proof. If $S_{i2}(\lambda) = \emptyset$, assume to the contrary that $\lambda^* \notin \{\lambda^2_{i-1}, \lambda_2, \lambda'_2\}$. Then, our previous two assumptions on λ^* are true and $\lambda^* \in (\lambda'_1, \lambda'_2)$. According to our algorithm invariants, $S_{i2}(\lambda^*) = S_{i2}(\lambda) = \emptyset$. This means that if we applied the decision algorithm on $\lambda = \lambda^*$, the sensor $s_{g(i)}(\lambda^*)$ would not exist. In other words, the decision algorithm would stop after the first $i-1$ steps, i.e., the decision algorithm would only use sensors in $S_{i-1}(\lambda^*)$ to cover all barriers.

On the other hand, according to our algorithm invariants, $R_{i-1}(\lambda) < \beta$ for all $\lambda \in (\lambda^1_{i-1}, \lambda^2_{i-1})$. Since $\lambda^* \in (\lambda'_1, \lambda'_2) \subseteq (\lambda^1_{i-1}, \lambda^2_{i-1})$, $R_{i-1}(\lambda^*) < \beta$, but this contradicts with that all barriers are covered by the sensors of $S_{i-1}(\lambda^*)$ after the first $i-1$ steps of the decision algorithm. □

By Lemma 5.5.3, if $S_{i2}(\lambda) = \emptyset$, then λ^* is the smallest feasible value of $\{\lambda^2_{i-1}, \lambda_2, \lambda'_2\}$, which can be found by performing three feasibility tests. Otherwise, we proceed as follows.

We make the third assumption that $\lambda^* \neq \lambda'_2$. Thus, $\lambda^* \in (\lambda'_1, \lambda'_2)$ and $S_{i2}(\lambda^*) = S_{i2}(\lambda)$ for any $\lambda \in (\lambda'_1, \lambda'_2)$. Next, we compute $s_{g(i)}(\lambda^*)$, i.e., the leftmost sensor of $S_{i2}(\lambda^*)$. Although $S_{i2}(\lambda)$ is the same for all $\lambda \in (\lambda'_1, \lambda'_2)$, the leftmost sensor of $S_{i2}(\lambda)$ may not be the same for all $\lambda \in (\lambda'_1, \lambda'_2)$. For each sensor $s_k \in S_{i2}(\lambda)$ and any

$\lambda \in (\lambda_1', \lambda_2')$, the location of s_k in the configuration $C_{i-1}(\lambda)$ is $x_k^r(\lambda)$. As discussed before, $x_k^r(\lambda)$ for $\lambda \in (\lambda_1', \lambda_2')$ defines a piece of the upper branch of a hyperbola in the 2D coordinate system in which the x-coordinates correspond to the λ values and the y-coordinates correspond to $x_k^r(\lambda)$ values. We consider the lower envelope $\mathcal{L}$ of the functions $x_k^r(\lambda)$ defined by all sensors s_k of $S_{i2}(\lambda)$. For each point q of $\mathcal{L}$, suppose q lies on the function defined by a sensor s_k and q's x-coordinate is λ_q. If $\lambda = \lambda_q$, then the leftmost sensor of $S_{i2}(\lambda)$ is s_k. This means that each curve segment of $\mathcal{L}$ defined by one sensor corresponds to the same leftmost sensor of $S_{i2}(\lambda)$. Based on this observation, we compute $s_{g(i)}(\lambda^*)$ as follows.

Since the functions $x_k^r(\lambda)$ and $x_j^r(\lambda)$ of two sensors s_k and s_j have at most one intersection in (λ_1', λ_2'), the number of vertices of the lower envelope $\mathcal{L}$ is $O(n)$ and $\mathcal{L}$ can be computed in $O(n \log n)$ time [68–70]. Let Λ be the set of the x-coordinates of the vertices of $\mathcal{L}$. We also add λ_1' and λ_2' to Λ. After sorting all values of Λ, by using our decision algorithm, we do binary search on the sorted list of Λ to find two adjacent values λ_1'' and λ_2'' such that $\lambda^* \in (\lambda_1'', \lambda_2'']$. Note that $(\lambda_1'', \lambda_2''] \subseteq (\lambda_1', \lambda_2']$. Since λ_1'' and λ_2'' are two adjacent values of the sorted Λ, by our above analysis, there is a sensor that is always the leftmost sensor of $S_{i2}(\lambda)$ for all $\lambda \in (\lambda_1'', \lambda_2'']$. To find the sensor, we can take any value λ in $(\lambda_1'', \lambda_2'')$ and explicitly compute the locations of sensors in $S_{i2}(\lambda)$. The above computes $s_{g(i)}(\lambda^*)$ in $O(m \log n + n \log n \log\log n)$ time.

Finally, we let $\lambda_i^1 = \lambda_1''$, $\lambda_i^2 = \lambda_2''$, and $S_i(\lambda) = S_{i-1}(\lambda) \cup \{s_{g(i)}(\lambda^*)\}$.

If the above computes λ^*, then we terminate the algorithm. Otherwise, we obtain an interval $(\lambda_i^1, \lambda_i^2] \subseteq (\lambda_{i-1}^1, \lambda_{i-1}^2]$ that contains λ^* and the set $S_i(\lambda)$. If $s_{g(i)}(\lambda) \in S_{i1}(\lambda)$, then $R_i(\lambda)$ is equal to $x_{g(i)} + \sqrt{\lambda^2 - y_{g(i)}^2} + r$. If $s_{g(i)}(\lambda) \in S_{i2}(\lambda)$, then $R_i(\lambda) = R_{i-1}(\lambda) + 2r$. By the third algorithm invariant, $R_i(\lambda)$ is either constant or equal to $x_j + \sqrt{\lambda^2 - y_j^2} + c'$ for some constant c' and some sensor s_j with $1 \le j \le i-1$

If it is not true that $R_i(\lambda) < \beta$ for all $\lambda \in (\lambda_i^1, \lambda_i^2)$, then we preform some additional processing as follows. We first have the following lemma.

Lemma 5.5.4. *If it is not true that $R_i(\lambda) < \beta$ for all $\lambda \in (\lambda_i^1, \lambda_i^2)$, then $R_i(\lambda)$ is strictly increasing on $(\lambda_i^1, \lambda_i^2)$ and there is a single value $\lambda' \in (\lambda_i^1, \lambda_i^2)$ such that $R_i(\lambda') = \beta$.*

Proof. The proof is almost the same as that of Lemma 4 in [61] and we include it here for the sake of completeness.

Since it is not true that $R_i(\lambda) < \beta$ for all $\lambda \in (\lambda_i^1, \lambda_i^2)$, either $R_i(\lambda) > \beta$ for all $\lambda \in (\lambda_i^1, \lambda_i^2)$, or there is a value $\lambda' \in (\lambda_i^1, \lambda_i^2)$ with $R_i(\lambda') = \beta$. We first argue that the former case cannot happen.

Assume to the contrary that $R_i(\lambda) > \beta$ for all $\lambda \in (\lambda_i^1, \lambda_i^2)$. Then, $R_i(\lambda') > \beta$ for any $\lambda' \in (\lambda_i^1, \lambda^*)$ since $\lambda^* \in (\lambda_i^1, \lambda_i^2]$. But this would imply that we have found a feasible solution using only sensors in $S_i(\lambda)$ and the maximum movement of all sensors in $S_i(\lambda)$ is at most $\lambda' < \lambda^*$, contradicting with that λ^* is the maximum moving distance in an optimal solution.

Hence, there is a value $\lambda' \in (\lambda_i^1, \lambda_i^2)$ with $R_i(\lambda') = \beta$. Next, we show that $R_i(\lambda)$ must be a strictly increasing function. Assume to the contrary this is not true. Then, $R_i(\lambda)$ must be constant on $(\lambda_i^1, \lambda_i^2)$. Thus, $R_i(\lambda) = \beta$ for all $\lambda \in (\lambda_i^1, \lambda_i^2)$. Since $\lambda^* \in (\lambda_i^1, \lambda_i^2]$, let λ' be any value in (λ_i^1, λ^*). Hence, $R_i(\lambda') = \beta$, and as above, λ' is a feasible value. However, $\lambda' < \lambda^*$ incurs contradiction. □

By Lemma 5.5.4, we compute the value $\lambda' \in (\lambda_i^1, \lambda_i^2)$ such that $R_i(\lambda') = \beta$. This means that all barriers are covered by the sensors of $S_i(\lambda')$ in $C_i(\lambda')$, and thus λ' is a feasible value and $\lambda^* \in (\lambda_i^1, \lambda']$. Because $R_i(\lambda)$ is strictly increasing, $R_i(\lambda) < \beta$ for all $\lambda \in (\lambda_i^1, \lambda')$. We update λ_i^2 to λ'.

In either case, $R_i(\lambda) < \beta$ now holds for all $\lambda \in (\lambda_i^1, \lambda_i^2)$. Finally, we perform the jump-over procedure, as follows.

If $R_i(\lambda)$ is a constant and $R_i(\lambda)$ is not in the interior of a barrier, then we set $R_i(\lambda)$ to the left endpoint of the next barrier. If $R_i(\lambda)$ is an increasing function, then we do the following. If we increase λ from λ_i^1 to λ_i^2, an *event* happens if $R_i(\lambda)$ is equal to the left or right endpoint of a barrier at some *event value* of λ. During the increasing of λ, between any two adjacent events, $R_i(\lambda)$ is either always in the interior of a barrier or is always between two barriers. We compute all event values in $O(m)$ time by using the function $R_i(\lambda)$ and the endpoints of all barriers. Let Λ denote the set of all event values, and we also add λ_i^1 and λ_i^2 to Λ. After sorting all values in Λ, using the decision algorithm in Lemma 5.5.2, we do binary search on the sorted list of Λ to find two adjacent values λ_1 and λ_2 such that $\lambda^* \in (\lambda_1, \lambda_2]$. Note that $(\lambda_1, \lambda_2] \subseteq (\lambda_i^1, \lambda_i^2]$. Since $|\Lambda| = O(m)$, the binary search calls the decision algorithm $O(\log m)$ times, which takes overall $O(m \log m + n \log\log n \log m)$ time. Finally, we reset $\lambda_i^1 = \lambda_1$ and $\lambda_i^2 = \lambda_2$.

This completes the i-th step of the algorithm, which runs in $O((m + n \log\log n) \cdot (\log m + \log n))$ time. If λ^* is not computed in this step, then it can be verified that all algorithm variants are maintained (the analysis is similar to that in [61], so we omit it). The algorithm will compute λ^* after at most n steps. The total time of the algorithm is $O(n \cdot (m + n \log\log n) \cdot (\log m + \log n))$, which is bounded by $O(nm \log m + n^2 \log n \log\log n)$ as shown in the following theorem. Note that the space of the algorithm is $O(n)$.

Theorem 5.5.5. *The problem MBC can be solved in $O(nm \log m + n^2 \log n \log\log n)$ time and $O(n)$ space.*

Proof. As discussed before, the running time of the algorithm is $O(n \cdot (m + n \log\log n) \cdot (\log m + \log n))$, which is $O(nm \log m + n^2 \log n \log\log n + nm \log n + n^2 \log m \log\log n)$. We claim that $nm \log n + n^2 \log m \log\log n = O(nm \log m + n^2 \log n \log\log n)$. Indeed, if $m \leq n \log\log n$, then $nm \log n = O(n^2 \log n \log\log n)$ and $n^2 \log m \log\log n = O(n^2 \log n \log\log n)$; otherwise, $nm \log n = O(nm \log m)$ and $n^2 \log m \log\log n = O(nm \log m)$. □

5.6 Concluding Remarks

As mentioned before, the high-level scheme of our algorithm for MBC is similar to those in [33, 61]. However, a new technique we propose in this chapter can help reduce the space complexities of the algorithms in [33, 61]. Specifically, Chen et al. [33] solved the line-constrained problem in $O(n^2 \log n)$ time and $O(n^2)$ space for the case where $m = 1$ and sensors have different ranges. Wang and Zhang [61] solved the line-constrained problem in $O(n^2 \log n \log\log n)$ time and $O(n^2)$ space for the case where $m = 1$, sensors have the same range, and sensors have weights. If we apply the similar preprocessing as in Lemma 5.5.1, then the space complexities of both algorithms [33,61] can be reduced to $O(n)$ while the time complexities do not change asymptotically.

In addition, by slightly changing our algorithm for MBC, we can also solve the following problem variant: Find a subset S' of sensors of S to move them to L to cover all barriers such that the maximum movement of all sensors of S' is minimized (and sensors of $S \setminus S'$ do not move). We omit the details.

CHAPTER 6

SEPARATING OVERLAPPED INTERVALS ON A LINE

6.1 Introduction

We consider the following *separating overlapped intervals* problem on a line in this chapter. The results in this chapter were submitted to a conference in 2018 and is now still under review.

6.2 Problem Definitions and Our Results

Let $\mathcal{I}$ be a set of n intervals on a real line ℓ. We say that two intervals *overlap* if their intersection contains more than one point. In this chapter, we consider an *interval separation problem*: move the intervals of $\mathcal{I}$ on ℓ such that no two intervals overlap and the maximum moving distance of these intervals is minimized.

If all intervals of $\mathcal{I}$ have the same length, then after the left endpoints of the intervals are sorted, the problem can be solved in $O(n)$ time by an easy greedy algorithm [19]. For the general problem where intervals may have different lengths, to the best of our knowledge, the problem has not been studied before. In this chapter, we present an $O(n \log n)$ time and $O(n)$ space algorithm for it. We also show an $\Omega(n \log n)$ time lower bound for solving the problem under the algebraic decision tree model, and thus our algorithm is optimal.

As a basic problem and like many other interval problems, the interval separation problem potentially has many applications. For example, one possible application is on scheduling, as follows. Suppose there are n jobs that need to be completed on a machine. Each job requests a starting time and a total time for using the machine (hence it is a time interval). The machine can only work on one job at any time, and once it works on one job, it is not allowed to switch to other jobs until the job is finished. If the requested time intervals of the jobs have any overlap, then we have to change the requested starting times of some intervals. In order to minimize deviations from their requested time intervals, one scheduling strategy could be changing the requested starting times (either advance or delay) such that the maximum difference between the requested starting times and the scheduled starting times of all jobs is minimized. Clearly, the problem is an instance of the interval separation problem. The problem also has applications in the following scenario. Suppose a wireless sensor network has n wireless mobile devices on a line and each device has a transmission range. We want to move the devices along the line to eliminate the interference such that the maximum moving distance of the devices is minimized (e.g., to save the energy). This is also an instance of the interval separation problem.

6.2.1 Applications and Related Work

Many interval problems have been used to model scheduling problems. We give a few examples. Given n jobs, each job requests a time interval to use a machine. Suppose there is only one machine and the goal is to find a maximum number of jobs whose requested time intervals do not have any overlap (so that they can use the machine). The problem can be solved in $O(n \log n)$ time by an easy greedy algorithm [71]. Another related problem is to find a minimum number of machines such that all jobs can be completed [71]. Garey et al. [6] studied a scheduling problem, which is essentially the following problem. Given n intervals on a line, determine whether it is possible to find a unit-length sub-interval in each input interval, such that no two sub-intervals overlap. An $O(n \log n)$ time algorithm was given in [6] for it. An optimization version of the problem was also studied [55, 56], where the goal is to find a maximum number of intervals that contain non-overlapping unit-length sub-intervals. Other scheduling problems on intervals have also been considered, e.g., see [5, 6, 8–11, 71].

Many problems on wireless sensor networks are also modeled as interval problems. For example, a mobile sensor barrier coverage problem can be modeled as the following interval problem. Given on a line n intervals (each interval is the region covered by a sensor at the center of the interval) and another segment B (called "barrier"), the goal is to move the intervals such that the union of the intervals fully covers B and the maximum moving distance of all intervals is minimized. If all intervals have the same length, Czyzowicz et al. [60] solved the problem in $O(n^2)$ time and later Chen et al. [33] improved it to $O(n \log n)$ time. If intervals have different lengths, Chen et al. [33] solved the problem in $O(n^2 \log n)$ time. The min-sum version of the problem has also been considered. If intervals have the same length, Czyzowicz et al. [62] gave an $O(n^2)$ time algorithm, and Andrews and Wang [63] solved the problem in $O(n \log n)$ time. If intervals have different lengths, then the problem becomes NP-hard [33]. Refer to [12–17] for other interval problems on mobile sensor barrier coverage.

Our interval separation problem may also be considered as a coverage problem in the sense that we want to move intervals of $\mathcal{I}$ to cover a total of maximum length of the line ℓ such that the maximum moving distance of the intervals is minimized.

6.2.2 Our Approach

We consider a *one-direction* version of the problem in which intervals of $\mathcal{I}$ are only allowed to move rightwards. We show (in Section 6.3) that the original "two-direction" problem can be reduced to the one-direction problem in the following way: If OPT is an optimal solution of the one-direction problem and δ_{opt} is the maximum moving distance of all intervals in OPT, then we can obtain an optimal solution for the two-direction problem by moving each interval in OPT leftwards by $\delta_{opt}/2$.

Hence, it is sufficient to solve the one-direction problem. It turns out that the difficulty is mainly on determining the order of intervals of $\mathcal{I}$ in OPT. Indeed, once such an "optimal order" is known, it is quite straightforward to compute the positions of the intervals in OPT in additional $O(n)$ time (i.e., consider the intervals in the order one by one and put each interval "as left as possible"). If all intervals have the same length, then such an optimal order is obvious, which is the order of the intervals sorted by their left endpoints in the input. Indeed, this is how the $O(n)$ time algorithm in [19] works.

However, if the intervals have different lengths, which is the case we consider in this chapter, then determining an optimal order is substantially more challenging. At first glance, it seems that we have to consider all possible orders of the intervals, whose number is exponential. By several interesting (and even surprising) observations, we show that we only need to consider at most n ordered lists of intervals. Consequently, a straightforward algorithm can find and maintain these "candidate" lists in $O(n^2)$ time and space. We call it the "preliminary algorithm", which is essentially a greedy algorithm. The algorithm is relatively simple but it is quite involved to prove its correctness. To this end, we extensively use the "exchange argument", which is a standard technique for proving correctness of greedy algorithms (e.g., see [71]).

To further improve the preliminary algorithm, we discover more observations, which help us "prune" some "redundant" candidate lists. More importantly, the remaining lists have certain monotonicity properties such that we are able to implicitly compute and maintain them in $O(n \log n)$ time and $O(n)$ space, although the number of the lists can still be $\Omega(n)$. Although the correctness analysis is fairly complicated, the algorithm is still quite simple and easy to implement (indeed, the most "complicated" data structure is a binary search tree).

The rest of the chapter is organized as follows. In Section 6.3, we give notation and reduce our problem to the one-direction case. In Section 6.4, we give our preliminary algorithm, whose correctness is proved in Section 6.5. The improved algorithm is presented in Section 6.6. In Section 6.7, we conclude the chapter and prove the $\Omega(n \log n)$ time lower bound by a reduction from the integer element distinctness problem [72, 73].

6.3 Preliminaries

We assume the line ℓ is the x-axis. The *one-direction* version of the interval separation problem is to move intervals of $\mathcal{I}$ on ℓ in one direction (without loss of generality, we assume it is the right direction) such that no two intervals overlap and the maximum moving distance of the intervals is minimized. Let OPT denote an optimal solution of the one-direction version and let δ_{opt} be the maximum moving distance of all intervals in OPT. The following lemma gives a reduction from the general "two-direction" problem to the one-direction problem.

Lemma 6.3.1. *An optimal solution for the interval separation problem can be obtained by moving every interval in OPT leftwards by $\delta_{opt}/2$.*

Proof. Let SOL be the solution obtained by moving every interval in OPT leftwards by $\delta_{opt}/2$. Our goal is to show that SOL is an optimal solution for our original problem. Let

δ be the maximum moving distance of all intervals in SOL. Since no intervals in OPT have been moved leftwards (with respect to their input positions), we have $\delta = \delta_{opt}/2$.

Assume to the contrary that SOL is not optimal. Then, there exists another solution SOL' for the original problem in which the maximum interval moving distance is $\delta' < \delta$. By moving every interval of SOL' rightwards by δ', we can obtain a feasible solution SOL'' for the one-direction problem in which no interval has been moved leftwards (with respect to their input positions) and the maximum interval moving distance of SOL'' is at most $2\delta'$, which is smaller than δ_{opt} since $\delta' < \delta$. However, this contradicts with that OPT is an optimal solution for the one-direction case. □ □

By Lemma 6.3.1, once we have an optimal solution for the one-direction problem, we can obtain an optimal solution for our original problem in additional $O(n)$ time. In the following, we will focus on solving the one-direction case.

We first sort all intervals of $\mathcal{I}$ by their left endpoints. For ease of exposition, we assume no two intervals have their left endpoints located at the same position (otherwise we could break ties by also sorting their right endpoints). Let $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ be the sorted intervals by their left endpoints from left to right. For each (integer) $i \in [1, n]$, denote by l_i and r_i the (physical) left and right endpoints of I_i, respectively. Denote by x_i^l and x_i^r the x-coordinates of l_i and r_i in the input, respectively. Note that for each $i \in [1, n]$, the two physical endpoints l_i and r_i may be moved during the algorithm, but the two coordinates x_i^l and x_i^r are always fixed. Denote by $|I_i|$ the length of I_i, i.e., $|I_i| = x_i^r - x_i^l$.

For convenience, when we say the *position* of an interval, we refer to the position of the left endpoint of the interval.

With respect to a subset $\mathcal{I}'$ of $\mathcal{I}$, by a *configuration* of $\mathcal{I}'$, we refer to a specification of the position of each interval of $\mathcal{I}'$. For example, in the input configuration of $\mathcal{I}$, interval I_i is at x_i^l for each $i \in [1, n]$. Given a configuration $\mathcal{C}$ of $\mathcal{I}'$, for each interval $I_i \in \mathcal{I}'$, if l_i is at x in $\mathcal{C}$, then we call the value $x - x_i^l$ the *displacement* of I_i, denoted by $d(i, \mathcal{C})$, and if $d(i, \mathcal{C}) \geq 0$, then we say that I_i is *valid* in $\mathcal{C}$. We say that $\mathcal{C}$ is *feasible* if the displacement of every interval of $\mathcal{I}'$ is valid and no two intervals of $\mathcal{I}'$ overlap in $\mathcal{C}$. The maximum displacement of the intervals of $\mathcal{I}'$ in $\mathcal{C}$ is called the *max-displacement* of $\mathcal{C}$, denoted by

$\delta(\mathcal{C})$. Hence, finding an optimal solution for the one-direction problem is equivalent to computing a feasible configuration of $\mathcal{I}$ whose max-displacement is minimized; such a configuration is also called an *optimal configuration*.

For convenience of discussion, depending on the context, we will use the intervals I_i of $\mathcal{I}$ and their indices i interchangeably. For example, $\mathcal{I}$ may also refer to the set of indices $\{1, 2, \ldots, n\}$.

Let L_{opt} be the list of intervals of $\mathcal{I}$ in an optimal configuration sorted from left to right. We call L_{opt} an *optimal list*. Given L_{opt}, we can compute an optimal configuration in $O(n)$ time by an easy greedy algorithm, called the *left-possible placement strategy*: Consider the intervals following their order in L_{opt}, and for each interval, place it on ℓ as left as possible so that it does not overlap with the intervals that are already placed on ℓ and its displacement is non-negative. The following lemma formally gives the algorithm and proves its correctness.

Lemma 6.3.2. *Given an optimal list L_{opt}, we can compute an optimal configuration in $O(n)$ time by the left-possible placement strategy.*

Proof. We first describe the algorithm and then prove its correctness.

We consider the indices one by one following their order in L_{opt}. Consider any index i. If I_i is the first interval of L_{opt}, then we place I_i at x_i^l (i.e., I_i stays at its input position). Otherwise, let I_j be the previous interval of I_i in L_{opt}. So I_j has already been placed on ℓ. Let x be the current x-coordinate of the right endpoint r_j of I_j. We place the left endpoint l_i of I_i at $\max\{x_i^l, x\}$. If I_i is the last interval of L_{opt}, then we finish the algorithm. Clearly, the algorithm can be easily implemented in $O(n)$ time.

Let $\mathcal{C}$ be the configuration of all intervals obtained by the above algorithm. Recall that $\delta(\mathcal{C})$ denote the max-displacement of $\mathcal{C}$. Below, we show that $\mathcal{C}$ is an optimal configuration.

Indeed, since L_{opt} is an optimal list, there exists an optimal configuration $\mathcal{C}'$ in which the order of the indices of $\mathcal{I}$ follows that in L_{opt}. Hence, the max-displacement of $\mathcal{C}'$ is δ_{opt}. According to our greedy strategy for computing $\mathcal{C}$, it is not difficult to see that the position of each interval I_i of $\mathcal{I}$ in $\mathcal{C}$ cannot be strictly to the right of its position in

$\mathcal{C}'$. Therefore, the displacement of each interval in $\mathcal{C}$ is no larger than that in $\mathcal{C}'$. This implies that $\delta(\mathcal{C}) \leq \delta_{opt}$. Therefore, $\mathcal{C}$ is an optimal configuration. □ □

Due to Lemma 6.3.2, we will focus on computing an optimal list L_{opt}.

For any subset $\mathcal{I}'$ of $\mathcal{I}$, an *(ordered) list* of $\mathcal{I}'$ refers to a permutation of the indices of $\mathcal{I}'$. Let L be a list of $\mathcal{I}$ and let L' be a list of $\mathcal{I}'$ with $\mathcal{I}' \subseteq \mathcal{I}$. We say that L' is *consistent with* L if the relative order of indices of $\mathcal{I}'$ in L is the same as that in L'. If L' is consistent with an optimal list L_{opt} of $\mathcal{I}$, then we call L' a *canonical list* of $\mathcal{I}'$.

For any $1 \leq i \leq j \leq n$, we use $\mathcal{I}[i,j]$ to denote the subset of consecutive intervals of $\mathcal{I}$ from i to j, i.e, $\{i, i+1, \ldots, j\}$.

6.4 The Preliminary Algorithm

In this section, we describe an algorithm that can compute an optimal list in $O(n^2)$ time and space. The correctness of the algorithm is mainly discussed in Section 6.5.

Our algorithm considers the intervals of $\mathcal{I}$ one by one by their index order. After each interval I_i is processed, we obtain a set $\mathcal{L}$ of at most i lists of the indices of $\mathcal{I}[1,i]$, such that $\mathcal{L}$ contains at least one canonical list of $\mathcal{I}[1,i]$. For each list $L \in \mathcal{L}$, a feasible configuration $\mathcal{C}_L$ of the intervals of $\mathcal{I}[1,i]$ is also maintained. As will be clear later, $\mathcal{C}_L$ is essentially the configuration obtained by applying the left-possible placement strategy on the intervals of $\mathcal{I}[1,i]$ following their order in L. For each $j \in [1,i]$, we let $x_j^l(\mathcal{C}_L)$ and $x_j^r(\mathcal{C}_L)$ respectively denote the x-coordinates of l_j and r_j in $\mathcal{C}_L$ (recall that l_j and r_j are the left and right endpoints of the interval I_j, respectively). Recall that $\delta(\mathcal{C}_L)$ denotes the max-displacement of $\mathcal{C}_L$, i.e, the maximum displacement of the intervals of $\mathcal{I}[1,i]$ in $\mathcal{C}_L$.

Initially when $i = 1$, we have only one list $L = \{1\}$ and let $\mathcal{C}_L$ consist of the single interval I_1 at its input position, i.e., $x_1^l(\mathcal{C}_L) = x_1^l$. Clearly, $\delta(\mathcal{C}_L) = 0$. We let $\mathcal{L}$ consist of the only list L. It is vacuously true that L is a canonical list of $\mathcal{I}[1,1]$.

In general, assume interval I_{i-1} has been processed and we have the list set $\mathcal{L}$ as discussed above. In the following, we give our algorithm for processing I_i. Consider a list $L \in \mathcal{L}$. Note that $\mathcal{C}_L$ has been computed, which is a feasible configuration of $\mathcal{I}[1,i-1]$. The value $\delta(\mathcal{C}_L)$ is also maintained. Let m be the last index in L. Note that

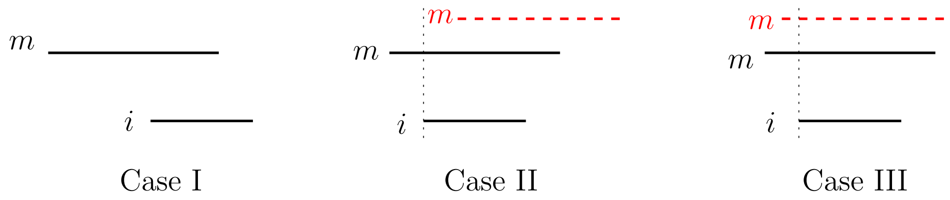


Figure 6.1. Illustrating the three main cases. The (black) solid segments show intervals in their input positions and the (red) dashed segments shows interval I_m in $\mathcal{C}_L$.

$m < i$. Depending on the values of x_i^l, x_i^r, x_m^r, and $x_m^l(\mathcal{C}_L)$, there are three main cases (e.g. see Fig. 6.1).

Case I: $x_i^r \geq x_m^r$ (i.e., the right endpoint r_i of I_i is to the right of r_m in the input).. In this case, we update L by appending i to the end of L. Further, we update the configuration $\mathcal{C}_L$ by placing l_i at $\max\{x_m^r(\mathcal{C}_L), x_i^l\}$ (which follows the left-possible placement strategy). We let L' denote the original list of L before i is inserted and let $\mathcal{C}_{L'}$ denote the original configuration of $\mathcal{C}_L$. We update $\delta(\mathcal{C}_L)$ by the following observation.

Observation 6.4.1. *$\mathcal{C}_L$ is a feasible configuration and* $\delta(\mathcal{C}_L) = \max\{\delta(\mathcal{C}_{L'}), x_i^l(\mathcal{C}_L) - x_i^l\}$.

Proof. By our way of setting I_i in $\mathcal{C}_L$, I_i is valid and does not overlap with any other interval in $\mathcal{C}_L$. Hence, $\mathcal{C}_L$ is feasible. Comparing with $\mathcal{C}_{L'}$, $\mathcal{C}_L$ has one more interval I_i. Therefore, $\delta(\mathcal{C}_L)$ is equal to the larger value of $\delta(\mathcal{C}_{L'})$ and the displacement of I_i in $\mathcal{C}_L$, which is $x_i^l(\mathcal{C}_L) - x_i^l$. □

The following lemma will be used to show the correctness of our algorithm and its proof is deferred to Section 6.5.

Lemma 6.4.2. *If L' is a canonical list of $\mathcal{I}[1, i-1]$, then L is a canonical list of $\mathcal{I}[1, i]$.*

Case II: $x_i^r < x_m^r$ and $x_i^l \leq x_m^l(\mathcal{C}_L)$.. In this case, we update L by inserting i right before m. Let $x = x_m^l(\mathcal{C}_L)$. We update $\mathcal{C}_L$ by setting l_i at x and setting l_m at $x + |I_i|$. We let L' denote the original list of L before inserting i and let $\mathcal{C}_{L'}$ denote the original $\mathcal{C}_L$. We update $\delta(\mathcal{C}_L)$ by the following observation. Note that $x_m^l(\mathcal{C}_L)$ now refers to the position of l_m in the updated $\mathcal{C}_L$.

Observation 6.4.3. *$\mathcal{C}_L$ is a feasible configuration and* $\delta(\mathcal{C}_L) = \max\{\delta(\mathcal{C}_{L'}), x_m^l(\mathcal{C}_L) - x_m^l\}$.

Proof. Since $x_i^l \leq x$ and l_i is at x in $\mathcal{C}_L$, I_i is valid in $\mathcal{C}_L$. Comparing with its position in $\mathcal{C}_{L'}$, I_m has been moved rightwards; since I_m is valid in $\mathcal{C}_{L'}$, I_m is also valid in $\mathcal{C}_L$. Note that no two intervals overlap in $\mathcal{C}_L$. Therefore, $\mathcal{C}_L$ is a feasible configuration.

Comparing with $\mathcal{C}_{L'}$, $\mathcal{C}_L$ has one more interval I_i and I_m has been moved rightwards in $\mathcal{C}_L$. Therefore, $\delta(\mathcal{C}_L)$ is equal to the maximum of the following three values: $\delta(\mathcal{C}_{L'})$, the displacement of I_i in $\mathcal{C}_L$, and the displacement of I_m in $\mathcal{C}_L$. Observe that the displacement of I_i is smaller than that of I_m. This is because l_m is to the left of l_i in the input (since $m < i$) while l_m is to the right of l_i in $\mathcal{C}_L$. Thus, it holds that $\delta(\mathcal{C}_L) = \max\{\delta(\mathcal{C}_{L'}), x_m^l(\mathcal{C}_L) - x_m^l\}$. □

The proof of the following lemma is deferred to Section 6.5.

Lemma 6.4.4. *If L' is a canonical list of $\mathcal{I}[1, i-1]$, then L is a canonical list of $\mathcal{I}[1, i]$.*

Case III: $x_i^r < x_m^r$ and $x_i^l > x_m^l(\mathcal{C}_L)$.. In this case, we first update L by appending i to the end of L and update $\mathcal{C}_L$ by placing the left endpoint of I_i at $x_m^r(\mathcal{C}_L)$. Let L' be the original list L before we insert i and let $\mathcal{C}_{L'}$ be the original configuration of $\mathcal{C}_L$.

Further, we create a new list L^*, which is the same as L except that we switch the order of i and m. Thus, m is the last index of L^*. Correspondingly, the configuration $\mathcal{C}_{L^*}$ is the same as $\mathcal{C}_L$ except that l_i is at x_i^l, i.e., its position in the input, and l_m is at x_i^r. We say that L^* is the *new list generated* by L'. We do not put L^* in the set $\mathcal{L}$ at this moment (but L is in $\mathcal{L}$).

Observation 6.4.5. *Both $\mathcal{C}_L$ and $\mathcal{C}_{L^*}$ are feasible; $\delta(\mathcal{C}_L) = \max\{\delta(\mathcal{C}_{L'}), x_i^l(\mathcal{C}_L) - x_i^l\}$ and $\delta(\mathcal{C}_{L^*}) = \max\{\delta(\mathcal{C}_{L'}), x_m^l(\mathcal{C}_{L^*}) - x_m^l\}$.*

Proof. By a similar argument as in Observation 6.4.1, $\mathcal{C}_L$ is feasible and $\delta(\mathcal{C}_L) = \max\{\delta(\mathcal{C}_{L'}), x_i^l(\mathcal{C}_L) - x_i^l\}$. By a similar argument as in Observation 6.4.3, $\mathcal{C}_{L^*}$ is feasible and $\delta(\mathcal{C}_{L^*}) = \max\{\delta(\mathcal{C}_{L'}), x_m^l(\mathcal{C}_{L^*}) - x_m^l\}$. We omit the details. □

The proof of the following lemma is deferred to Section 6.5.

Lemma 6.4.6. *If L' is a canonical list of $\mathcal{I}[1, i-1]$, then one of L and L^* is a canonical list of $\mathcal{I}[1, i]$.*

After each list L of $\mathcal{L}$ is processed as above, let $\mathcal{L}^*$ denote the set of all new generated lists in Case III. Recall that no list of $\mathcal{L}^*$ has been added into $\mathcal{L}$ yet. Let L^*_{min} be the list of $\mathcal{L}^*$ with the minimum value $\delta(\mathcal{C}_{L^*_{min}})$. The proof of the following lemma is deferred to Section 6.5.

Lemma 6.4.7. *If $\mathcal{L}^*$ has a canonical list of $\mathcal{I}[1,i]$, then L^*_{min} is a canonical list of $\mathcal{I}[1,i]$.*

Due to Lemma 6.4.7, among all lists of $\mathcal{L}^*$, we only need to keep L^*_{min}. So we add L^*_{min} to $\mathcal{L}$ and ignore all other lists of $\mathcal{L}^*$. We call L^*_{min} a *new list* of $\mathcal{L}$ produced by our algorithm for processing I_i and all other lists of $\mathcal{L}$ are considered as the *old lists*.

Remark.. Lemma 6.4.7 is a key observation that helps avoid maintaining an exponential number of lists.

This finishes our algorithm for processing the interval I_i. Clearly, $\mathcal{L}$ has at most one more new list. After I_n is processed, the list L of $\mathcal{L}$ with minimum $\delta(\mathcal{C}_L)$ is an optimal list.

According to our above description, the algorithm can be easily implemented in $O(n^2)$ time and space. The proof of Theorem 6.4.8 gives the details and also shows the correctness of the algorithm based on Lemmas 6.4.2, 6.4.4, 6.4.6, and 6.4.7.

Theorem 6.4.8. *An optimal solution for the one-direction problem can be found in $O(n^2)$ time and space.*

Proof. To implement the algorithm, we can use a linked list to represent each list of $\mathcal{L}$. Consider a general step for processing interval I_i.

For any list $L \in \mathcal{L}$, inserting i to L can be easily done in $O(1)$ time for each of the three cases. The configuration $\mathcal{C}_L$ and the value $\delta(\mathcal{C}_L)$ can also be updated in $O(1)$ time. If L generates a new list L^*, then we do not explicitly construct L^* but only compute the value $\delta(\mathcal{C}_{L^*})$, which can be done in $O(1)$ time by Observation 6.4.5. Once every list $L \in \mathcal{L}$ has been processed, we find the list $L^*_{min} \in \mathcal{L}^*$. Then, we explicitly construct L^* and $\mathcal{C}_{L^*}$, in $O(n)$ time.

Hence, each general step for processing I_i can be done in $O(n)$ time since $\mathcal{L}$ has at most n lists. Thus, the total time and space of the algorithm is $O(n^2)$.

For the correctness, after a general step for processing I_i, Lemmas 6.4.2, 6.4.4, 6.4.6, and 6.4.7 together guarantee that the set $\mathcal{L}$ has at least one canonical list of $\mathcal{I}[1,i]$. After

$j = 6$ ———

——— $k = 10$

$$L_{opt} : \cdots\cdots, k = 10, 8, 14, 5, 4, 12, j = 6, \cdots\cdots$$

$$L^1_{opt}[j,k] = \{8, 5, 4\} \qquad L^2_{opt}[j,k] = \{k = 10, 14, 12\}$$

$$L'_{opt} : \cdots\cdots, 8, 5, 4, j = 6, 14, k = 10, 12, \cdots\cdots$$

Figure 6.2. Illustrating an inversion (j, k) of L_{opt} and an example for Lemma 6.5.1: the intervals j and k are shown in their input positions.

I_n is processed, since $\mathcal{C}_L$ is essentially obtained by the left-possible placement strategy for each list $L \in \mathcal{L}$, if L is the list of $\mathcal{L}$ with the smallest $\delta(\mathcal{C}_L)$, then L is an optimal list and $\mathcal{C}_L$ is an optimal configuration by Lemma 6.3.2. □

6.5 The Correctness of the Preliminary Algorithm

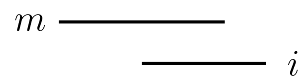
In this section, we establish the correctness of our preliminary algorithm. Specifically, we will prove Lemmas 6.4.2, 6.4.4, 6.4.6, and 6.4.7. The major analysis technique is the exchange argument, which is quite standard for proving correctness of greedy algorithms (e.g., see [71]).

Let L be a list of all indices of $\mathcal{I}$. For any two indices $j, k \in [1, n]$, let $L[j, k]$ denote the sub-list of all indices of L between j and k (including j and k).

For any $1 \leq j < k \leq n$, we say that (j, k) is an *inversion* of L if $x^r_j \leq x^r_k$ and k is before j in L (k and j are not necessarily consecutive in L; e.g., see Fig. 6.2 with $L = L_{opt}$). For an inversion (j, k), we further introduce two sets of indices $L^1[j, k]$ and $L^2[j, k]$ as follows (e.g., see Fig. 6.2 with $L = L_{opt}$). Let $L^1[j, k]$ consist of all indices $i \in L[j, k]$ such that $i < k$ and $i \neq j$; let $L^2[j, k]$ consist of all indices $i \in L[j, k]$ such that $i \geq k$. Hence, $L^1[j, k]$, $L^2[j, k]$, and $\{j\}$ form a partition of the indices of $L[j, k]$.

We first give the following lemma, which will be extensively used later.

Lemma 6.5.1. *Let L_{opt} be an optimal list of all indices of $\mathcal{I}$. If L_{opt} has an inversion (j, k), then there exists another optimal list L'_{opt} that is the same as L_{opt} except that the sublist $L_{opt}[j, k]$ is changed to the following: all indices of $L^1_{opt}[j, k]$ are before j and all indices of $L^2_{opt}[j, k]$ are after j (in particular, k is after j, so (j, k) is not an inversion any more in L'_{opt}), and further, the relative order of the indices of $L^1_{opt}[j, k]$ in L'_{opt} is the same as that in L_{opt} (but this may not be the case for $L^2_{opt}[j, k]$). E.g., see Fig. 6.2.*



$$L_{opt} : \cdots\cdots, i, \cdots, m, \cdots\cdots$$

$$L'_{opt} : \cdots\cdots, L^1_{opt}[i, m], m, \cdots, i, \cdots\cdots$$

Figure 6.3. Illustrating the proof of Lemma 6.4.2. The intervals m and i are shown in their input positions.

Many proofs given later in the chapter will utilize Lemma 6.5.1 as a basic technique for "eliminating" inversions in optimal lists. Before giving the proof of Lemma 6.5.1, which is somewhat technical, lengthy, and tedious, we first show that Lemma 6.4.2 can be easily proved with the help of Lemma 6.5.1.

6.5.1 Proof of Lemma 6.4.2.

Assume L' is a canonical list of $\mathcal{I}[1, i-1]$. Our goal is to prove that L is a canonical list of $\mathcal{I}[1, i]$.

Since L' is a canonical list, by the definition of a canonical list, there exists an optimal configuration $\mathcal{C}$ in which the order of the intervals of $\mathcal{I}[1, i-1]$ is the same as that in L'. Let L_{opt} be the list of indices of the intervals of $\mathcal{I}$ in $\mathcal{C}$. If i is after m in L_{opt}, then L is consistent with L_{opt} and thus is a canonical list of $\mathcal{I}[1, i]$. In the following, we assume i is before m in L_{opt}.

Since $m < i$, $x^r_m \leq x^r_i$, and i is before m in L_{opt}, (m, i) is an inversion in L_{opt}. Let L'_{opt} be another optimal list obtained by applying Lemma 6.5.1 on (m, i). Refer to Fig. 6.3. We claim that L is consistent with L'_{opt}, which will prove that L is a canonical list. We prove the claim below.

Indeed, note that L' is consistent with L_{opt}. Comparing with L_{opt}, by Lemma 6.5.1, only the indices of the sublist $L_{opt}[m, i]$ have their relative order changed in L'_{opt}. Since all indices of L' are smaller than i, by definition, all indices of L' that are in $L_{opt}[m, i]$ are contained in $L^1_{opt}[m, i]$. By Lemma 6.5.1, the relative order of the indices of $L^1_{opt}[m, i]$ in L'_{opt} is the same as that in L_{opt}, and further, all indices of $L^1_{opt}[m, i]$ are still before m in L'_{opt}. This implies that the relative order of the indices of L' does not change from L_{opt} to L'_{opt}. Hence, L' is consistent with L'_{opt}. On the other hand, by Lemma 6.5.1, i is after m. Thus, L is consistent with L'_{opt}. This proves the claim and thus proves

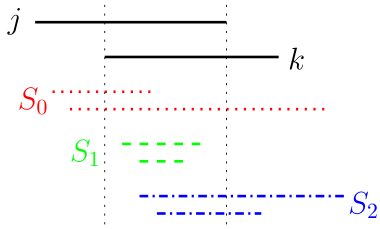


Figure 6.4. Illustrating the intervals of $L_{opt}[j,k]$ in their input positions. The two (red) dotted intervals are in $S_0 = L^1_{opt}[j,k]$; the two (green) dashed intervals are in S_1; the two (blue) dashed-dotted intervals are in S_2.

Lemma 6.4.2.

6.5.2 Proof of Lemma 6.5.1

In this section, we give the proof of Lemma 6.5.1.

We partition the set $L^2_{opt}[j,k] \setminus \{k\}$ into two sets S_1 and S_2, defined as follows (e.g., see Fig. 6.4). Let S_1 consists of all indices t of $L^2_{opt}[j,k] \setminus \{k\}$ such that $x^r_t \le x^r_j$ (i.e., r_t is to the left of r_j in the input). Let S_2 consists of all indices of $L^2_{opt}[j,k] \setminus \{k\}$ that are not in S_1. Note that $L_{opt}[j,k] = L^1_{opt}[j,k] \cup S_1 \cup S_2 \cup \{j,k\}$. To simplify the notation, let $S = L_{opt}[j,k]$ and $S_0 = L^1_{opt}[j,k]$ (e.g., see Fig. 6.4).

We only consider the general case where none of S_0, S_1, and S_2 is empty since other cases can be analyzed by similar but simpler techniques.

In the following, from L_{opt}, we will subsequently construct a sequence of optimal lists L_0, L_1, L_2, L_3, such that eventually L_3 is the list L'_{opt} specified in the statement of Lemma 6.5.1 (e.g., see Fig. 6.5).

The List L_0

For any adjacent indices h and g of $L_{opt}[j,k] \setminus \{j,k\}$ such that h is before g in L_{opt}, we say that (h,g) is an *exchangeable pair* if one of the three cases happen: $g \in S_0$ and $h \in S_1$; $g \in S_1$ and $h \in S_2$; $g \in S_0$ and $h \in S_2$.

In the following, we will perform certain "exchange operations" to eliminate all exchangeable pairs of L_{opt}, after which we will obtain another optimal list L_0 in which for any $i_0 \in S_0$, $i_1 \in S_1$, $i_2 \in S_2$, i_0 is before i_1 and i_2 is after i_1, and all other indices of L_0 have the same positions as in L_{opt} (e.g., see Fig. 6.5).

$$L_0 : \cdots, k, S_0, S_1, S_2, j, \cdots$$
$$L_1 : \cdots, S_0, k, S_1, S_2, j, \cdots$$
$$L_2 : \cdots, S_0, k, S_1, j, S_2, \cdots$$
$$L_3 : \cdots, S_0, j, S_1, k, S_2, \cdots$$

Figure 6.5. Illustrating the relative order of k, j, S_0, S_1, S_2 in the four lists $L_0, L_1, L_2.L_3$.

Consider any exchangeable pair (h, g) of L_{opt}. Let L' be another list that is the same as L_{opt} except that h and g exchange their order. We call this an *exchange operation*. In the following, we show that L' is an optimal list.

Since L_{opt} is an optimal list, there is an optimal configuration $\mathcal{C}$ in which the order of the intervals is the same as L_{opt}. Consider the configuration $\mathcal{C}'$ that is the same as $\mathcal{C}$ except that we exchange the order of h and g in the following way (e.g., see Fig 6.6): $x_g^l(\mathcal{C}') = x_h^l(\mathcal{C})$ and $x_h^r(\mathcal{C}') = x_g^r(\mathcal{C})$, i.e., the left endpoint l_g of I_g in $\mathcal{C}'$ is at the same position as l_h in $\mathcal{C}$ and the right end point r_h of I_h in $\mathcal{C}'$ is at the same position as r_g in $\mathcal{C}$. Clearly, the order of intervals in $\mathcal{C}'$ is the same as that in L'. In the following, we show that $\mathcal{C}'$ is an optimal configuration, which will prove that L' is an optimal list.

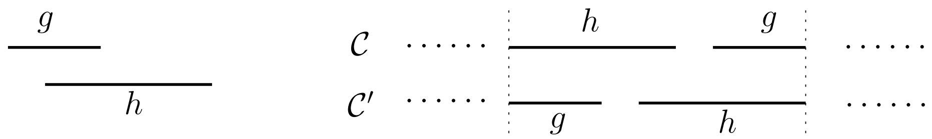


Figure 6.6. Left: Illustrating the intervals g and h at their input positions. Right: Illustrating the two intervals h and g in the configurations $\mathcal{C}$ and $\mathcal{C}'$ (note that h and g do not have to be connected).

We first show that $\mathcal{C}'$ is feasible. Recall that intervals h and g are adjacent in L_{opt} and also in L'. By our way of setting I_g and I_h in $\mathcal{C}'$, the segments of ℓ "spanned" by I_h and I_g in both $\mathcal{C}$ and $\mathcal{C}'$ are exactly the same (e.g., the segments between the two vertical dotted lines in Fig. 6.6). Since no two intervals of $\mathcal{I}$ overlap in $\mathcal{C}$, no two intervals overlap in $\mathcal{C}'$ as well.

Next, we show that every interval of $\mathcal{I}$ is valid in $\mathcal{C}'$. To this end, it is sufficient to show that I_h and I_g are valid in $\mathcal{C}'$ since other intervals do not change positions from $\mathcal{C}$ to $\mathcal{C}'$. For I_h, comparing with its position in $\mathcal{C}$, I_h has been moved rightwards in $\mathcal{C}'$, and thus I_h is valid in $\mathcal{C}'$. For I_g, since (h, g) is an exchangeable pair, g is either in S_0 or in S_1. In either case, $x_g^l \leq x_k^r$. On the other hand, I_k is to the left of I_g in $\mathcal{C}'$, which implies that $x_k^r(\mathcal{C}') \leq x_g^l(\mathcal{C}')$. Since I_k does not change position from $\mathcal{C}$ to $\mathcal{C}'$ and I_k

is valid in $\mathcal{C}$, we have $x_k^r \leq x_k^r(\mathcal{C}) = x_k^r(\mathcal{C}')$. Combining the above discussion, we have $x_g^l \leq x_k^r \leq x_k^r(\mathcal{C}) = x_k^r(\mathcal{C}') \leq x_g^l(\mathcal{C}')$. Thus, I_g is valid in $\mathcal{C}'$. This proves that $\mathcal{C}'$ is a feasible configuration.

We proceed to show that $\mathcal{C}'$ is an optimal configuration by proving that the max-displacement of $\mathcal{C}'$ is no more than the max-displacement of $\mathcal{C}$, i.e., $\delta(\mathcal{C}') \leq \delta(\mathcal{C})$. Note that $\delta(\mathcal{C}) = \delta_{opt}$ since $\mathcal{C}$ is an optimal configuration. Comparing with $\mathcal{C}$, I_g has been moved leftwards and I_h has been moved rightwards in $\mathcal{C}'$. Therefore, to prove $\delta(\mathcal{C}') \leq \delta_{opt}$, it suffices to show that the displacement of I_h in $\mathcal{C}'$, i.e., $d(h, \mathcal{C}')$, is at most δ_{opt}. Since (h, g) is an exchangeable pair, h is either in S_1 or in S_2. In either case, $x_j^l \leq x_h^l$. On the other hand, I_j is to the right of I_h in $\mathcal{C}'$, which implies that $x_h^l(\mathcal{C}') \leq x_j^l(\mathcal{C}')$. Consequently, we have $d(h, \mathcal{C}') = x_h^l(\mathcal{C}') - x_h^l \leq x_j^l(\mathcal{C}') - x_j^l = d(j, \mathcal{C}')$. Since I_j does not change position from $\mathcal{C}$ to $\mathcal{C}'$, $d(h, \mathcal{C}') \leq d(j, \mathcal{C}') = d(j, \mathcal{C}) \leq \delta_{opt}$. This proves that $\mathcal{C}'$ is an optimal configuration and L' is an optimal list.

If L' still has an exchangeable pair, then we keep applying the above exchange operations until we obtain an optimal list L_0 that does not have any exchangeable pairs. Hence, L_0 has the following property: for any $i_t \in S_t$ for $t = 0, 1, 2$, i_0 is before i_1 and i_2 is after i_1, and all other indices of L_0 have the same positions as in L_{opt}. Further, notice that our exchange operation never changes the relative order of any two indices in S_t for each $0 \leq t \leq 2$. In particular, the relative order of the indices of S_0 in L_{opt} is the same as that in L_0.

The List L_1

Let L_1 be another list that is the same as L_0 except that k is between the indices of S_0 and the indices of S_1 (e.g., see Fig. 6.5). In the following, we show that L_1 is also an optimal list. This can be done by keeping performing exchange operations between k and its right neighbor in S_0 until all indices of S_0 are to the left of k. The details are given below.

Let g be the right neighboring index of k in L_0 and g is in S_0. Let L' be the list that is the same as L_0 except that we exchange the order of k and g. In the following, we show that L' is an optimal list.

Since L_0 is an optimal list, there is an optimal configuration $\mathcal{C}$ in which the order

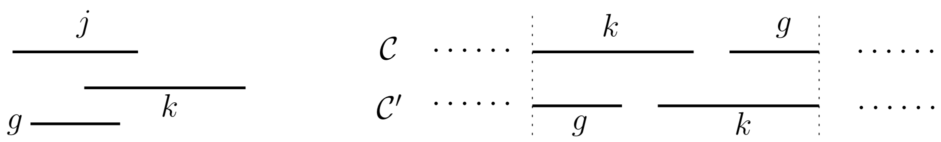


Figure 6.7. Left: Illustrating the intervals j, k, and g at their input positions. Right: Illustrating the two intervals k and g in the configurations $\mathcal{C}$ and $\mathcal{C}'$.

of the indices of the intervals is the same as L_0. Consider the configuration $\mathcal{C}'$ that is the same as $\mathcal{C}$ except that we exchange the order of k and g in the following way: $x_g^l(\mathcal{C}') = x_k^l(\mathcal{C})$ and $x_k^r(\mathcal{C}') = x_g^r(\mathcal{C})$ (e.g., see Fig. 6.7; similar to that in Section 6.5.2). In the following, we show that $\mathcal{C}'$ is an optimal solution, which will prove that L' is an optimal list.

We first show that $\mathcal{C}'$ is feasible. By the similar argument as in Section 6.5.2, no two intervals overlap in $\mathcal{C}'$. Next we show that every interval is valid in $\mathcal{C}$. It is sufficient to show that both I_k and I_g are valid. For I_k, comparing with its position in $\mathcal{C}$, I_k has been moved rightwards in $\mathcal{C}'$ and thus I_k is valid in $\mathcal{C}'$. For I_g, since $g \in S_0$, by the definition of S_0, $x_g^l \leq x_k^l$ (e.g., see the left side of Fig. 6.7). Since $x_k^l \leq x_k^l(\mathcal{C}) = x_g^l(\mathcal{C}')$, we obtain that $x_g^l \leq x_g^l(\mathcal{C}')$ and I_g is valid in $\mathcal{C}'$.

We proceed to show that $\mathcal{C}'$ is an optimal configuration by proving that $\delta(\mathcal{C}') \leq \delta(\mathcal{C}) = \delta_{opt}$. Comparing with $\mathcal{C}$, I_g has been moved leftwards and I_k has been moved rightwards in $\mathcal{C}'$. Therefore, to prove $\delta(\mathcal{C}') \leq \delta_{opt}$, it suffices to show that $d(k, \mathcal{C}') \leq \delta_{opt}$. Recall that l_j is to the left of l_k in the input. Note that k is to the left of j in L'. Hence, l_k is to the left of l_j in $\mathcal{C}'$. Thus, $d(k, \mathcal{C}') \leq d(j, \mathcal{C}')$. Note that $d(j, \mathcal{C}') = d(j, \mathcal{C})$ since the position of I_j does not change from $\mathcal{C}$ to $\mathcal{C}'$. Therefore, we obtain $d(k, \mathcal{C}') \leq d(j, \mathcal{C}) \leq \delta_{opt}$. This proves that $\mathcal{C}'$ is an optimal configuration and L' is an optimal list.

If the right neighbor of k in L' is still in S_0, then we keep performing the above exchange until all indices of S_0 are to the left of k, at which moment we obtain the list L_1. Thus, L_1 is an optimal list.

The List L_2

Let L_2 be another list that is the same as L_1 except that j is between the indices of S_1 and the indices of S_2 (e.g., see Fig. 6.5). This can be done by keeping performing exchange operations between j and its left neighbor in S_2 until all indices of S_2 are to

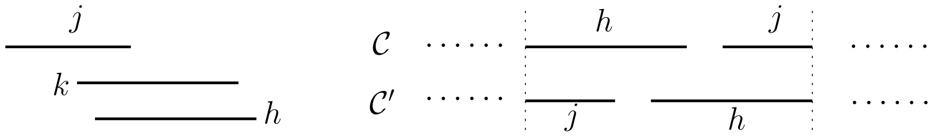


Figure 6.8. Left: Illustrating the intervals j, k, and h at their input positions. Right: Illustrating the two intervals h and j in the configurations $\mathcal{C}$ and $\mathcal{C}'$.

the right of j, which is symmetric to that in Section 6.5.2. The details are given below.

Let h be the left neighbor of j in L_1 and h is in S_2. Let L' be the list that is the same as L_1 except that we exchange the order of h and j. In the following, we show that L' is an optimal list.

Since L_1 is an optimal list, there is an optimal configuration $\mathcal{C}$ in which the order of the indices of the intervals is the same as L_1. Consider the configuration $\mathcal{C}'$ that is the same as $\mathcal{C}$ except that we exchange the order of j and h in the following way: $x_j^l(\mathcal{C}') = x_h^l(\mathcal{C})$ and $x_h^r(\mathcal{C}') = x_j^r(\mathcal{C})$ (e.g., see Fig. 6.8). In the following, we show that $\mathcal{C}'$ is an optimal solution, which will prove that L' is an optimal list.

We first show that $\mathcal{C}'$ is feasible. By the similar argument as before, no two intervals overlap in $\mathcal{C}'$. Next we show that every interval is valid in $\mathcal{C}'$. It is sufficient to show that both I_j and I_h are valid. For I_h, comparing with its position in $\mathcal{C}$, I_h has been moved rightwards in $\mathcal{C}'$ and thus I_h is valid in $\mathcal{C}'$. For I_j, since $h \in S_2$, by the definition of S_2, $x_j^l \leq x_h^l$. Since $x_h^l \leq x_h^l(\mathcal{C}) = x_j^l(\mathcal{C}')$, we obtain that $x_j^l \leq x_j^l(\mathcal{C}')$ and I_j is valid in $\mathcal{C}'$.

We proceed to show that $\mathcal{C}'$ is an optimal configuration by proving that $\delta(\mathcal{C}') \leq \delta(\mathcal{C}) = \delta_{opt}$. Comparing with $\mathcal{C}$, I_j has been moved leftwards and I_h has been moved rightwards in $\mathcal{C}'$. Therefore, to prove $\delta(\mathcal{C}') \leq \delta_{opt}$, it suffices to show that $d(h, \mathcal{C}') \leq \delta_{opt}$. Since h is in S_2, $x_j^r \leq x_h^r$. Since $x_h^r(\mathcal{C}') = x_j^r(\mathcal{C})$, we deduce $d(h, \mathcal{C}') = x_h^r(\mathcal{C}') - x_h^r \leq x_j^r(\mathcal{C}) - x_j^r = d(j, \mathcal{C}) \leq \delta_{opt}$. This proves that $\mathcal{C}'$ is an optimal configuration and L' is an optimal list.

If the left neighbor of j in L' is still in S_2, then we keep performing the above exchange until all indices of S_2 are to the right of j, at which moment we obtain the list L_2. Thus, L_2 is an optimal list.

The List L_3

Let L_3 be the list that is the same as L_2 except that we exchange the order of k

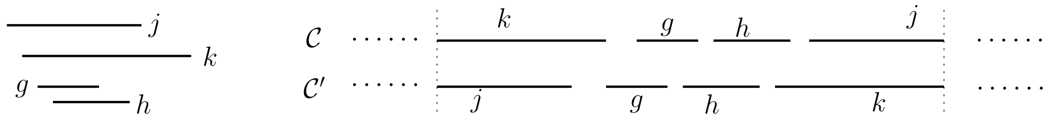


Figure 6.9. Left: Illustrating the intervals j, k, g and h at their input positions, where $S_1 = \{g, h\}$. Right: Illustrating the intervals of $S_1 \cup \{j, k\}$ in the configurations $\mathcal{C}$ and $\mathcal{C}'$.

and j, i.e., in L_3, the indices of S_1 are all after j and before k (e.g., see Fig. 6.5). In the following, we prove that L_3 is an optimal list.

Since L_2 is an optimal list, there is an optimal configuration $\mathcal{C}$ in which the order of the indices of intervals is the same as L_2. Consider the configuration $\mathcal{C}'$ that is the same as $\mathcal{C}$ except the following (e.g., see Fig. 6.9): First, we set $x_j^l(\mathcal{C}') = x_k^l(\mathcal{C})$; second, we shift each interval of S_1 leftwards by distance $|I_k| - |I_j|$ (if this value is negative, we actually shift rightwards by its absolute value); third, we set $x_k^r(\mathcal{C}') = x_j^r(\mathcal{C})$ (i.e., r_k is at the same position as r_j in $\mathcal{C}$). Clearly, the interval order of $\mathcal{C}'$ is the same as L_3. In the following, we show that $\mathcal{C}'$ is an optimal configuration, which will prove that L_3 is an optimal list.

We first show that $\mathcal{C}'$ is feasible. By our way of setting positions of intervals in $S_1 \cup \{j, k\}$, One can easily verify that no two intervals of $\mathcal{C}'$ overlap. Next we show that every interval is valid in $\mathcal{C}'$. It is sufficient to show that all intervals in $S_1 \cup \{j, k\}$ are valid. Comparing with $\mathcal{C}$, I_k has been moved rightwards in $\mathcal{C}'$. Thus, I_k is valid in $\mathcal{C}'$. Recall that $x_j^l \le x_k^l$ and $x_j^l(\mathcal{C}') = x_k^l(\mathcal{C})$. Since $x_k^l \le x_k^l(\mathcal{C})$ (because I_k is valid in $\mathcal{C}$), we obtain that $x_j^l \le x_j^l(\mathcal{C}')$ and I_j is valid in $\mathcal{C}'$. Consider any index $t \in S_1$. By the definition of S_1, $x_t^l \le x_j^r$. Since j is to the left of t in $\mathcal{C}'$, we have $x_j^r(\mathcal{C}') \le x_t^l(\mathcal{C}')$. Since $x_j^r \le x_j^r(\mathcal{C}')$ (because I_j is valid in $\mathcal{C}'$), we obtain that $x_t^l \le x_j^r \le x_j^r(\mathcal{C}') \le x_t^l(\mathcal{C}')$ and thus I_t is valid in $\mathcal{C}'$. This proves that $\mathcal{C}'$ is feasible.

We proceed to show that $\mathcal{C}'$ is an optimal configuration by proving that $\delta(\mathcal{C}') \le \delta(\mathcal{C}) = \delta_{opt}$. It is sufficient to show that for any $t \in S_1 \cup \{j, k\}$, $d(t, \mathcal{C}') \le \delta_{opt}$. Comparing with $\mathcal{C}$, I_j has been moved leftwards in $\mathcal{C}'$, and thus, $d(j, \mathcal{C}') \le d(j, \mathcal{C}) \le \delta_{opt}$. Recall that $x_j^r \le x_k^r$ and $x_k^r(\mathcal{C}') = x_j^r(\mathcal{C})$. We can deduce $d(k, \mathcal{C}') = x_k^r(\mathcal{C}') - x_k^r \le x_j^r(\mathcal{C}) - x_j^r \le d(j, \mathcal{C}) \le \delta_{opt}$. Consider any $t \in S_1$. By the definition of S_1, $x_t^l \ge x_k^l$. On the other hand, since t is to the left of k in $\mathcal{C}'$, $x_t^l(\mathcal{C}') \le x_k^l(\mathcal{C}')$. Therefore, we obtain that $d(t, \mathcal{C}') = x_t^l(\mathcal{C}') - x_t^l \le x_k^l(\mathcal{C}') - x_k^l = d(k, \mathcal{C}')$. We have proved above that $d(k, \mathcal{C}') \le \delta_{opt}$, and thus $d(t, \mathcal{C}') \leq \delta_{opt}$. This proves that $\mathcal{C}'$ is an optimal configuration and L_3 is an optimal list.

Notice that L_3 is the list L'_{opt} specified in the lemma statement. Indeed, in all above lists from L_{opt} to L_3, the relative order of the indices of S_0 (which is $L^1_{opt}[j,k]$) never changes. This proves Lemma 6.5.1.

6.5.3 Proof of Lemma 6.4.4

In this section, we prove Lemma 6.4.4. Assume L' is a canonical list of $\mathcal{I}[1, i-1]$. Our goal is to prove that L is also a canonical list of $\mathcal{I}[1, i]$.

Since L' is a canonical list, there exists an optimal configuration $\mathcal{C}$ in which the order the intervals of $\mathcal{I}[1, i-1]$ is the same as that in L'. Let L_{opt} be the list of indices of the intervals of $\mathcal{I}$ in $\mathcal{C}$. If, in L_{opt}, i is before m and after every index of $\mathcal{I}[1, i-1] \setminus \{m\}$, then L is consistent with L_{opt} and thus is a canonical list of $\mathcal{I}[1, i]$, so we are done with the proof.

In the following, we assume L is not consistent with L_{opt}. There are two cases. In the first case, i is after m in L_{opt}. In the second case, i is before j in L_{opt} for some $j \in \mathcal{I}[1, i-1] \setminus \{m\}$. We analyze the two cases below. In each case, by performing certain exchange operations and using Lemma 6.5.1, we will find an optimal list of all intervals of $\mathcal{I}$ such that L is consistent with the list (this will prove that L is an canonical list of $\mathcal{I}[1, i]$).

The First Case

Assume i is after m in L_{opt}. Let S denote the set of indices strictly between m and i in L_{opt} (so neither m nor i is in S). Since all indices of $\mathcal{I}[1, i-1]$ are before m in L_{opt}, it holds that $j > i$ for each index $j \in S$. Let S' be the set of indices j of S such that $x^r_j \geq x^r_i$. Note that for each $j \in S'$, the pair (i, j) is an inversion. We consider the general case where neither S nor S' is empty since the analysis for other cases is similar but easier.

Let j be the rightmost index of S'. Again, (i, j) is an inversion. By Lemma 6.5.1, we can obtain another optimal list L'_{opt} such that j is after i and positions of the indices other than those in S are the same as before in L_{opt}. Further, the indices strictly between

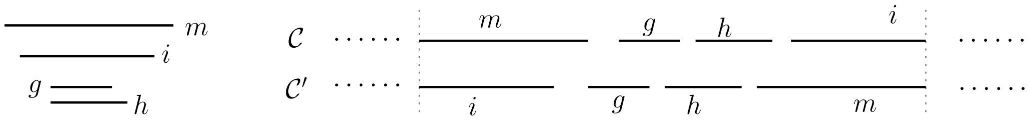


Figure 6.10. Left: Illustrating the intervals j, k, g and h at their input positions, where $S_0 = \{g, h\}$. Right: Illustrating the intervals of $S_0 \cup \{m, i\}$ in the configurations $\mathcal{C}$ and $\mathcal{C}'$.

m and i in L'_{opt} are all in S. If there is an index j between m and i in L'_{opt} such that (i, j) is an inversion, then we apply Lemma 6.5.1 again. We do this until we obtain an optimal list L_0 in which for any index j strictly between m and i, (i, j) is not an inversion, and thus $x^r_j < x^r_i$ (this further implies that I_j is contained in I_i in the input as $i < j$). Let S_0 denote the set of indices strictly between m and i in L_0.

Consider the list L_1 that is the same as L_0 except that we exchange the positions of m and i, i.e., the indices of S_0 are now after i and before m. In the following, we prove that L_1 is an optimal list. Note that L is consistent with L_1, and thus once we prove that L_1 is an optimal list, we also prove that L is a canonical list of $\mathcal{I}[1, i]$. The technique for proving that L_1 is an optimal list is similar to that in Section 6.5.2. The details are given below.

Since L_0 is an optimal list, there is an optimal configuration $\mathcal{C}$ in which the order of the indices of intervals is the same as L_0. Consider the configuration $\mathcal{C}'$ that is the same as $\mathcal{C}$ except the following (e.g., see Fig. 6.10): First, we set $x^l_i(\mathcal{C}') = x^l_m(\mathcal{C})$; second, we shift each interval of S_0 leftwards by distance $|I_m| - |I_i|$ (again, if this value is negative, we actually shift rightwards by its absolute value); third, we set $x^r_m(\mathcal{C}') = x^r_i(\mathcal{C})$. Clearly, the interval order in $\mathcal{C}'$ is the same as L_1. In the following, we show that $\mathcal{C}'$ is an optimal configuration, which will prove that L_1 is an optimal list.

We first show that $\mathcal{C}'$ is feasible. As in Section 6.5.2, no two intervals of $\mathcal{C}'$ overlap. Next, we show that every interval is valid in $\mathcal{C}'$. It is sufficient to show that all intervals in $S_0 \cup \{m, i\}$ are valid since other intervals do no change positions from $\mathcal{C}$ to $\mathcal{C}'$. Comparing with its position in $\mathcal{C}$, I_m has been moved rightwards in $\mathcal{C}'$. Thus, I_m is valid in $\mathcal{C}'$. Recall that in Case II of our algorithm, it holds that $x^l_i \leq x^l_m(\mathcal{C}_{L'})$, where $\mathcal{C}_{L'}$ is the configuration of only the intervals of $\mathcal{I}[1, i-1]$ following their order in L'. Since $\mathcal{C}_{L'}$ is the configuration constructed by the left-possible placement strategy and the order of the indices of $\mathcal{I}[1, i-1]$ in $\mathcal{C}$ is the same as L', it holds that $x^l_m(\mathcal{C}_{L'}) \leq x^l_m(\mathcal{C})$. Hence, we obtain $x_i^l \leq x_m^l(\mathcal{C})$. Since $x_i^l(\mathcal{C}') = x_m^l(\mathcal{C})$, $x_i^l \leq x_i^l(\mathcal{C}')$ and I_i is valid in $\mathcal{C}'$. Consider any index $j \in S_0$. Recall that I_j is contained in I_i in the input. Thus, $x_j^l \leq x_i^r$. Since i is to the left of j in $\mathcal{C}'$, we have $x_i^r(\mathcal{C}') \leq x_j^l(\mathcal{C}')$. Since $x_i^r \leq x_i^r(\mathcal{C}')$ (because I_i is valid in $\mathcal{C}'$), we obtain that $x_j^l \leq x_j^l(\mathcal{C}')$ and I_j is valid in $\mathcal{C}'$. This proves that $\mathcal{C}'$ is feasible.

We proceed to show that $\mathcal{C}'$ is an optimal configuration by proving that $\delta(\mathcal{C}') \leq \delta(\mathcal{C}) = \delta_{opt}$. It suffices to show that for any $j \in S_0 \cup \{m, i\}$, $d(j, \mathcal{C}') \leq \delta_{opt}$. Comparing with $\mathcal{C}$, I_i has been moved leftwards in $\mathcal{C}'$, and thus $d(i, \mathcal{C}') \leq d(i, \mathcal{C}) \leq \delta_{opt}$. Since $x_i^r \leq x_m^r$ and $x_m^r(\mathcal{C}') = x_i^r(\mathcal{C})$, we can deduce $d(m, \mathcal{C}') = x_m^r(\mathcal{C}') - x_m^r \leq x_i^r(\mathcal{C}) - x_i^r = d(i, \mathcal{C}) \leq \delta_{opt}$. Consider any $j \in S_0$. Recall that $x_j^l \geq x_i^l \geq x_m^l$. On the other hand, since j is to the left of m in $\mathcal{C}'$, $x_j^l(\mathcal{C}') \leq x_m^l(\mathcal{C}')$. Therefore, $d(j, \mathcal{C}') = x_j^l(\mathcal{C}') - x_j^l \leq x_m^l(\mathcal{C}') - x_m^l = d(m, \mathcal{C}')$. We have proved above that $d(m, \mathcal{C}') \leq \delta_{opt}$, and thus $d(j, \mathcal{C}') \leq \delta_{opt}$.

This proves that $\mathcal{C}'$ is an optimal configuration and L_1 is an optimal list. As discussed above, this also proves that L is a canonical list of $\mathcal{I}[1, i]$. This finishes the proof of the lemma in the first case.

The Second Case

In the second case, i is before j in L_{opt} for some $j \in \mathcal{I}[1, i-1] \setminus \{m\}$. We assume there is no other indices of $\mathcal{I}[1, i-1]$ strictly between i and j in L_{opt} (otherwise, we take j as the leftmost such index to the right of i).

Let $\widehat{L_0}$ be the list of indices of $\mathcal{I}[1, i]$ following their order in L_{opt}. Therefore, $\widehat{L_0}$ is a canonical list. Let $\widehat{L_1}$ be the list the same as $\widehat{L_0}$ except that the order of i and j is exchanged. In the following, we first show that $\widehat{L_1}$ is also a canonical list of $\mathcal{I}[1, i]$. The proof technique is very similar to the above first case.

Let S denote the set of indices strictly between i and j in L_{opt}. By the definition of j, $k > i > j$ holds for each index $k \in S$. Let S' be the set of indices k of S such that $x_k^r \geq x_j^r$. Hence, for each $k \in S'$, the pair (j, k) is an inversion of L_{opt}. We consider the general case where neither S nor S' is empty (otherwise the proof is similar but easier).

As in Section 6.5.3, starting from the rightmost index of S', we keep applying Lemma 6.5.1 to the inversion pairs and eventually obtain an optimal list L_0 in which for any index k of L_0 strictly between i and j, (j, k) is not an inversion and thus $x_k^r < x_j^r$

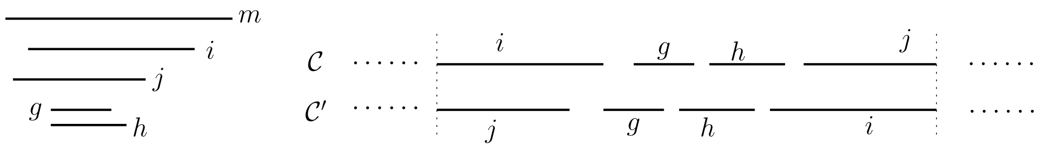


Figure 6.11. Left: Illustrating five intervals at their input positions, where $S_0 = \{g, h\}$. Right: Illustrating the intervals of $S_0 \cup \{i, j\}$ in the configurations $\mathcal{C}$ and $\mathcal{C}'$.

(hence $I_k \subseteq I_j$ in the input as $j < k$). Let S_0 denote the set of indices strictly between i and j in L_0.

Consider the list L_1 that is the same as L_0 except that we exchange the positions of i and j, i.e., the indices of S_0 are now after j and before i. In the following, we prove that L_1 is an optimal list, which will also prove that $\widehat{L_1}$ is a canonical list of $\mathcal{I}[1, i]$ since $\widehat{L_1}$ is consistent with L_1.

Since L_0 is an optimal list, there is an optimal configuration $\mathcal{C}$ in which the order of the intervals is the same as L_0. Consider the configuration $\mathcal{C}'$ that is the same as $\mathcal{C}$ except the following (e.g., see Fig. 6.11): First, we set $x_j^l(\mathcal{C}') = x_i^l(\mathcal{C})$; second, we shift each interval of S_0 leftwards by distance $|I_i| - |I_j|$; third, we set $x_i^r(\mathcal{C}') = x_j^r(\mathcal{C})$. Clearly, the interval order of $\mathcal{C}'$ is the same as L_1. Below, we show that $\mathcal{C}'$ is an optimal configuration, which will prove that L_1 is an optimal list.

We first show that $\mathcal{C}'$ is feasible. As before, no two intervals of $\mathcal{C}'$ overlap. Next we prove that all intervals in $S_0 \cup \{i, j\}$ are valid in $\mathcal{C}'$. Comparing with its position in $\mathcal{C}$, I_i has been moved rightwards in $\mathcal{C}'$ and thus is valid. Since $j < i$, $x_j^l < x_i^l$. Since $x_j^l(\mathcal{C}') = x_i^l(\mathcal{C})$ and $x_i^l \leq x_i^l(\mathcal{C})$ (because I_i is valid in $\mathcal{C}$), we obtain $x_j^l \leq x_j^l(\mathcal{C}')$ and I_j is valid in $\mathcal{C}'$. Consider any index $k \in S_0$. Recall that $x_k^l \leq x_k^r \leq x_j^r$. Since k is to the right of j in $\mathcal{C}'$, we have $x_j^r(\mathcal{C}') \leq x_k^l(\mathcal{C}')$. Since $x_j^r \leq x_j^r(\mathcal{C}')$, we obtain that $x_k^l \leq x_k^l(\mathcal{C}')$ and I_k is valid in $\mathcal{C}'$. This proves that $\mathcal{C}'$ is feasible.

We proceed to show that $\mathcal{C}'$ is an optimal configuration by proving that for any $k \in S_0 \cup \{i, j\}$, $d(k, \mathcal{C}') \leq \delta(\mathcal{C}) = \delta_{opt}$. Comparing with $\mathcal{C}$, I_j has been moved leftwards in $\mathcal{C}'$, and thus $d(j, \mathcal{C}') \leq d(j, \mathcal{C}) \leq \delta_{opt}$. Since $m < i$, l_m is to the left of r_i in the input. Since I_m is to the right of I_i in $\mathcal{C}'$, l_m is to the right of r_i in $\mathcal{C}'$. This implies that $d(i, \mathcal{C}') \leq d(m, \mathcal{C}')$. Since I_m does not change position from $\mathcal{C}$ to $\mathcal{C}'$, $d(m, \mathcal{C}') = d(m, \mathcal{C}) \leq \delta_{opt}$. Thus, we obtain $d(i, \mathcal{C}') \leq \delta_{opt}$. Consider any $k \in S_0$. Since $i < k$, $x_i^l \leq x_k^l$. On the other hand, since k is to the left of i in $\mathcal{C}'$, $x_k^l(\mathcal{C}') \leq x_i^l(\mathcal{C}')$. Therefore, we deduce

$d(k, \mathcal{C}') = x_k^l(\mathcal{C}') - x_k^l \leq x_i^l(\mathcal{C}') - x_i^l = d(i, \mathcal{C}')$. We have proved above that $d(i, \mathcal{C}') \leq \delta_{opt}$, and thus $d(k, \mathcal{C}') \leq \delta_{opt}$.

This proves that $\mathcal{C}'$ is an optimal configuration and L_1 is an optimal list. As discussed above, this also proves that $\widehat{L_1}$ is a canonical list of $\mathcal{I}[1, i]$.

If the right neighbor j of i in $\widehat{L_1}$ is not m, then by the same analysis as above, we can show that the list obtained by exchanging the order of i and j is still a canonical list of $\mathcal{I}[1, i]$. We keep applying the above exchange operation until we obtain a canonical list $\widehat{L_2}$ of $\mathcal{I}[1, i]$ such that the right neighbor of i in $\widehat{L_2}$ is m. Note that $\widehat{L_2}$ is exactly L, and thus this proves that L is a canonical list of $\mathcal{I}[1, i]$. This finishes the proof for the lemma in the second case.

Lemma 6.4.4 is thus proved.

6.5.4 Proof of Lemma 6.4.6

We prove Lemma 6.4.6. Assume that L' is a canonical list of $\mathcal{I}[1, i-1]$. Our goal is to prove that either L or L^* is a canonical list of $\mathcal{I}[1, i]$.

As L' is a canonical list, there exists an optimal list L_{opt} of $\mathcal{I}$ whose interval order is consistent with L'. Let $\widehat{L_0}$ be the list of indices of $\mathcal{I}[1, i]$ following the same order in L_{opt}. If $\widehat{L_0}$ is either L or L^*, then we are done with the proof. Otherwise, i must be before j in $\widehat{L_0}$ for some index $j \in \mathcal{I}[1, i-1] \setminus \{m\}$. By using the same proof as in Section 6.5.3, we can show that L^* is a canonical list of $\mathcal{I}[1, i]$. We omit the details.

6.5.5 Proof of Lemma 6.4.7

In this section, we prove Lemma 6.4.7. Assume $\mathcal{L}^*$ has a canonical list L_0 of $\mathcal{I}[1, i]$. Recall that L^*_{min} is the list of $\mathcal{L}^*$ with the smallest max-displacement. Our goal is to prove that L^*_{min} is also a canonical list of $\mathcal{I}[1, i]$.

Recall that for each list $L \in \mathcal{L}^*$, i and m are the last two indices with m at the end, and further, in the configuration $\mathcal{C}_L$ (which is obtained by the left-possible placement strategy on the intervals in $\mathcal{I}[1, i]$ following their order in L), $x_i^l(\mathcal{C}_L) = x_i^l$ and $x_m^l(\mathcal{C}_L) = x_i^r$. Also, each list of $\mathcal{L}^*$ is generated in Case III of the algorithm and we have $I_i \subseteq I_m$ in the input.

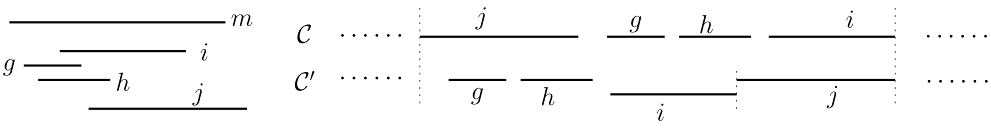


Figure 6.12. Left: Illustrating five intervals at their input positions, where $L_{opt}[j,i] = \{j,g,h,i\}$. Right: Illustrating the intervals of $L_{opt}[j,i]$ in the configurations $\mathcal{C}$ and $\mathcal{C}'$. (Interval i is shifted downwards in order to visually separate it from interval j.)

Since L_0 is a canonical list of $\mathcal{I}[1,i]$, there is an optimal list L_{opt} of $\mathcal{I}$ that is consistent with L_0. Let S be the set of indices of $\mathcal{I}[i+1,n]$ before i in L_{opt}. We consider the general case where S is not empty (otherwise the proof is similar but easier). Let j be the rightmost index of S in L_{opt}. Let L'_{opt} be the list that is the same as L_{opt} except that we move j right after i. In the following, we show that L'_{opt} is also an optimal list.

Since L_{opt} is an optimal list, there is an optimal configuration $\mathcal{C}$ in which the order of the indices of intervals is the same as L_{opt}. Recall that $L_{opt}[j,i]$ is consists of indices of L_{opt} between j and i inclusively. Consider the configuration $\mathcal{C}'$ that is the same as $\mathcal{C}$ except the following (e.g., see Fig. 6.12): First, for each index $k \in L_{opt}[j,i] \setminus \{j\}$, move I_k leftwards by distance $|I_j|$; second, move I_j rightwards such that l_j is at r_i (after I_i is moved leftwards in the above first step, so that I_i is connected with I_j). Note that the order of intervals of $\mathcal{I}$ in $\mathcal{C}'$ is exactly L'_{opt}. In the following, we show that $\mathcal{C}'$ is an optimal configuration, which will also prove that L'_{opt} is an optimal list.

We first show that $\mathcal{C}'$ is feasible. By our way of setting the positions of intervals in $L_{opt}[j,i]$, no two intervals overlap in $\mathcal{C}'$. Next, we show that every interval is valid in $\mathcal{C}'$. It is sufficient to show that I_k is valid in $\mathcal{C}'$ for every index k in $L_{opt}[j,i]$ since all other intervals do not move from $\mathcal{C}$ to $\mathcal{C}'$. Comparing with its position in $\mathcal{C}$, I_j has been moved rightwards in $\mathcal{C}'$ and thus is valid. Suppose $k \neq j$. By the definition of j, $k < j$ and thus $x^l_k \leq x^l_j$. By our way of constructing $\mathcal{C}'$, $x^l_j(\mathcal{C}) \leq x^l_k(\mathcal{C}')$. Since I_j is valid in $\mathcal{C}$, it holds that $x^l_j \leq x^l_j(\mathcal{C})$. Thus, we obtain that $x^l_k \leq x^l_k(\mathcal{C}')$ and I_k is valid. This proves that $\mathcal{C}'$ is feasible.

We proceed to show that $\mathcal{C}'$ is an optimal configuration by proving that $\delta(\mathcal{C}') \leq \delta(\mathcal{C}) = \delta_{opt}$. It is sufficient to show that for any index $k \in L_{opt}[j,i]$, $d(k,\mathcal{C}') \leq \delta_{opt}$. If k is not j, then comparing with $\mathcal{C}$, I_k has been moved leftwards, and thus $d(k,\mathcal{C}') \leq d(k,\mathcal{C}) \leq \delta_{opt}$. In the following, we show that $d(j,\mathcal{C}') \leq \delta_{opt}$. Indeed, since $m < i < j$, it holds that $x^l_m \leq x^l_j$. On the other hand, I_m is to the right of I_j in $\mathcal{C}'$, and thus,

$x_j^l(\mathcal{C}') \leq x_m^l(\mathcal{C}')$. Therefore, we have $d(j,\mathcal{C}') = x_j^l(\mathcal{C}') - x_j^l \leq x_m^l(\mathcal{C}') - x_m^l = d(m,\mathcal{C}')$. Since the position of I_m is the same in $\mathcal{C}$ and $\mathcal{C}'$, $d(m,\mathcal{C}') = d(m,\mathcal{C}) \leq \delta_{opt}$. Thus, we have $d(j,\mathcal{C}') \leq \delta_{opt}$. This proves that $\mathcal{C}'$ is an optimal configuration and L'_{opt} is an optimal list.

If there are still indices of $\mathcal{I}[i+1,n]$ before i in L'_{opt}, then we keep applying the above exchange operations until we obtain an optimal list L''_{opt} that does not have any index of $\mathcal{I}[i+1,n]$ before i, and in other words, the indices of L''_{opt} before i are exactly those in $\mathcal{I}[1,i-1] \setminus \{m\}$.

Since L''_{opt} is an optimal list, there is an optimal configuration $\mathcal{C}''$ whose interval order is the same as L''_{opt}. Let $\mathcal{C}'''$ be a configuration that is the same as $\mathcal{C}''$ except the following: For each interval I_k with $k \in \mathcal{I}[1,i-1] \setminus \{m\}$, we set its position the same as its position in $\mathcal{C}_{L^*_{min}}$ (which is the configuration obtained by our algorithm for the list L^*_{min}). Recall that the position of I_i in $\mathcal{C}_{L^*_{min}}$ is the same as that in the input. On the other hand, $x_i^l \leq x_i^l(\mathcal{C}'')$. Therefore, $\mathcal{C}'''$ is still a feasible configuration. We claim that $\mathcal{C}'''$ is also an optimal configuration. To see this, the maximum displacement of all intervals in $\mathcal{I}[1,i-1] \setminus \{m\}$ in $\mathcal{C}'''$ is at most $\delta(\mathcal{C}_{L^*_{min}})$. Recall that $\delta(\mathcal{C}_{L^*_{min}}) \leq \delta(\mathcal{C}_{L_0})$. Further, since L_0 is a canonical list, it holds that $\delta(\mathcal{C}_{L_0}) \leq \delta_{opt}$. Thus, we obtain $\delta(\mathcal{C}_{L^*_{min}}) \leq \delta_{opt}$. Consequently, the maximum displacement of all intervals in $\mathcal{I}[1,i-1] \setminus \{m\}$ in $\mathcal{C}'''$ is at most δ_{opt}. Since only intervals of $\mathcal{I}[1,i-1] \setminus \{m\}$ in $\mathcal{C}'''$ change positions from $\mathcal{C}''$ to $\mathcal{C}'''$, we obtain $\delta(\mathcal{C}''') \leq \delta_{opt}$ and thus $\mathcal{C}'''$ is an optimal configuration.

According to our construction of $\mathcal{C}'''$, the order of the intervals of $\mathcal{I}[1,i]$ in $\mathcal{C}'''$ is exactly L^*_{min}. Therefore, L^*_{min} is a canonical list of $\mathcal{I}[1,i]$. This proves Lemma 6.4.7.

6.6 The Improved Algorithm

In this section, we improve our preliminary algorithm to $O(n \log n)$ time and $O(n)$ space. The key idea is that based on new observations we are able to prune some "redundant" lists from $\mathcal{L}$ after each step of the algorithm (actually Lemma 6.4.7 already gives an example for pruning redundant lists). More importantly, although the number of remaining lists in $\mathcal{L}$ can still be $\Omega(n)$ in the worst case, the remaining lists of $\mathcal{L}$ have certain monotonicity properties such that we are able to implicitly maintain them in $O(n)$ space and update them in $O(\log n)$ amortized time for each step of the algorithm for processing an interval I_i.

In the following, we first give some observations that will help us to perform the pruning procedure on $\mathcal{L}$.

6.6.1 Observations

In this section, unless otherwise stated, let $\mathcal{L}$ be the set after a step of our preliminary algorithm for processing an interval i. Recall that for each list $L \in \mathcal{L}$, we also have a configuration $\mathcal{C}_L$ that is built following the left-possible placement strategy. We use $x(\mathcal{C}_L)$ to denote the x-coordinate of the right endpoint of the rightmost interval of L in $\mathcal{C}_L$.

For any two lists L_1 and L_2 of $\mathcal{L}$, we say that L_1 *dominates* L_2 if the following holds: If L_2 is a canonical list of $\mathcal{I}[1, i]$, then L_1 must also be a canonical list of $\mathcal{I}[1, i]$. Hence, if L_1 dominates L_2, then L_2 is "redundant" and can be pruned from $\mathcal{L}$.

The subsequent two lemmas give ways to identify redundant lists from $\mathcal{L}$. In general, Lemma 6.6.1 is for the case where two lists have different last indices while Lemma 6.6.2 is for the case where two lists have the same last index (notice the slight differences in the lemma conditions).

Lemma 6.6.1. *Suppose L_1 and L_2 are two lists of $\mathcal{L}$ such that the last index of L_1 is m', the last index of L_2 is m (with $m \neq m'$), and $x^r_{m'} \leq x^r_m$. Then, if $\delta(\mathcal{C}_{L_1}) \leq d(m, \mathcal{C}_{L_2})$ and $x(\mathcal{C}_{L_1}) \leq x(\mathcal{C}_{L_2})$, then L_1 dominates L_2.*

Proof. Assume L_2 is a canonical list of $\mathcal{I}[1, i]$. Our goal is to prove that L_1 is also a canonical list of $\mathcal{I}[1, i]$. It is sufficient to construct an optimal configuration in which the order the intervals of $\mathcal{I}[1, i]$ is L_1. We let h denote the left neighboring index of m' in L_1 and let g denote the left neighboring index of m in L_2.

Since L_2 is a canonical list, there is an optimal list Q that is consistent with L_2. Let S denote the set of indices of $\mathcal{I}[i + 1, n]$ before g in Q. We consider the general case where S is not empty (otherwise the proof is similar but easier).

By the similar analysis as in the proof of Lemma 6.4.7 (we omit the details), we can obtain an optimal list Q_1 that is the same as Q except that all indices of S are now right after g in Q_1 (i.e., all indices of Q before g except those in S are still before g in

$$Q_2 : \cdots\cdots g, S', m, k, \cdots\cdots$$
$$Q_3 : \cdots\cdots h, S', m', k, \cdots\cdots$$

Figure 6.13. Illustrating the two lists Q_2 and Q_3, where k is the right neighboring index of m in Q_2 and k is also right neighboring index of m' in Q_3. In Q_2 (resp., Q_3), the indices strictly before S' are exactly those in $\mathcal{I}[1,i] \setminus \{m\}$ (resp., $\mathcal{I}[1,i] \setminus \{m'\}$).

Q_1 with the same relative order, and all indices of Q after g are now after indices of S in Q_1 with the same relative order). Therefore, in Q_1, the indices before g are exactly those in $\mathcal{I}[1,i] \setminus \{m\}$.

Recall that $Q_1[g,m]$ denote the sublist of Q_1 between g and m including g and m. If there is an index j in $Q_1[g,m]$ such that (m,j) is an inversion, then as in the proof of Lemma 6.4.2, we keep applying Lemma 6.5.1 on all such indices j from right to left to obtain another optimal list Q_2 such that for each $j \in Q_2[g,m]$, (m,j) is not an inversion. Note that the indices before and including g in Q_1 are the same as those in Q_2. Let S' denote the set of indices of $Q_2[g,m] \setminus \{g,m\}$. Again, we consider the general case where S' is not empty. Note that $S' \subseteq \mathcal{I}[i+1,n]$. For each $j \in S'$, since (m,j) is not an inversion and $m < j$, it holds that $x_j^r < x_m^r$.

Let Q_3 be another list that is the same as Q_2 except the following (e.g., see Fig 6.13): First, we move m' right after the indices of S' and move m before the indices of S' (i.e., the indices of Q_3 from the beginning to m' are indices of $\mathcal{I}[1,i] \setminus \{m'\}$, indices of S', and m'); second, we re-arrange the indices of $\mathcal{I}[1,i] \setminus \{m'\}$ (which are all before indices of S' in Q_3) in exactly the same order as in L_1. In this way, L_1 is consistent with Q_3. In the following, we show that Q_3 is an optimal list, which will prove that L_1 is a canonical list of $\mathcal{I}[1,i]$ and thus prove the lemma.

Since Q_2 is an optimal list, there is an optimal configuration $\mathcal{C}_2$ whose interval order is Q_2. Consider the configuration $\mathcal{C}_3$ whose interval order follows Q_3 and whose interval positions are the same as those in $\mathcal{C}_2$ except the following: First, for each index $j \in \mathcal{I}[1,i] \setminus \{m'\}$, we set the position of I_j in the same as its position in $\mathcal{C}_{L_1}$ (i.e., the configuration obtained by our algorithm for L_1); second, we place the intervals of S' such that they do not overlap but connect together (i.e., the right endpoint co-locates with the left endpoint of the next interval) following their order in Q_2 and the left endpoint of the leftmost interval of S' is at the right endpoint of I_h (recall that h is the left neighbor of m' in L_1, which is also the rightmost interval of $\mathcal{I}[1,i] \setminus \{m'\}$ in Q_3; e.g., see Fig. 6.13); third, we set the left endpoint of $I_{m'}$ at the right endpoint of the rightmost interval of S'. Therefore, all intervals before and including m' do not have any overlap in $\mathcal{C}_3$, and the intervals of $S' \cup \{h, m'\}$ essentially connect together. In the following, we show that $\mathcal{C}_3$ is an optimal configuration, which will prove that Q_3 is an optimal list.

We first show that $\mathcal{C}_3$ is feasible. We begin with proving that no two intervals overlap. Let k be the right neighboring interval of m in Q_2 (e.g., see Fig. 6.13), and k now becomes the right neighboring interval of m' in Q_3. To prove no two intervals of $\mathcal{C}_3$ overlap, it is sufficient to show that $I_{m'}$ and I_k do not overlap, i.e., $x^r_{m'}(\mathcal{C}_3) \leq x^l_k(\mathcal{C}_3)$. Note that $x^l_k(\mathcal{C}_3) = x^l_k(\mathcal{C}_2)$ and $x^r_m(\mathcal{C}_2) \leq x^l_k(\mathcal{C}_2)$. Hence, it suffices to prove $x^r_{m'}(\mathcal{C}_3) \leq x^r_m(\mathcal{C}_2)$.

We claim that in the configuration $\mathcal{C}_{L_1}$, $l_{m'}$ is at r_h. Indeed, since $x^r_{m'} \leq x^r_m$ and I_m is to the left of $I_{m'}$ in $\mathcal{C}_{L_1}$, it holds that $x^l_{m'} \leq x^l_{m'}(\mathcal{C}_{L_1})$. Since $\mathcal{C}_{L_1}$ is constructed based on the left-possible placement strategy, we have $x^l_{m'}(\mathcal{C}_{L_1}) = x^r_h(\mathcal{C}_{L_1})$, which proves the claim.

Recall that by the definition of $x(\mathcal{C}_{L_1})$, we have $x(\mathcal{C}_{L_1}) = x^r_{m'}(\mathcal{C}_{L_1})$.

Let l be the total length of all intervals of S'. By our way of constructing $\mathcal{C}_3$, it holds that $x^r_{m'}(\mathcal{C}_3) = x^r_{m'}(\mathcal{C}_{L_1}) + l = x(\mathcal{C}_{L_1}) + l$. On the other hand, since L_2 is consistent with Q_2 and $\mathcal{C}_{L_2}$ is constructed based on the left-possible placement strategy, it holds that $x(\mathcal{C}_{L_2}) + l \leq x^r_m(\mathcal{C}_2)$. By the lemma condition, $x(\mathcal{C}_{L_1}) \leq x(\mathcal{C}_{L_2})$. Hence, we obtain $x^r_{m'}(\mathcal{C}_3) = x(\mathcal{C}_{L_1}) + l \leq x(\mathcal{C}_{L_2}) + l \leq x^r_m(\mathcal{C}_2)$. Thus, $I_{m'}$ and I_k do not overlap in $\mathcal{C}_3$.

We proceed to prove that every interval of $\mathcal{C}_3$ is valid. For any interval before h and including h in Q_3, since its position in $\mathcal{C}_3$ is the same as that in $\mathcal{C}_{L_1}$, it is valid. For interval m', since it is valid in $\mathcal{C}_{L_1}$ and $x^r_{m'}(\mathcal{C}_3) = x^r_{m'}(\mathcal{C}_{L_1}) + l$, it is also valid in $\mathcal{C}_3$. Consider any interval $j \in S'$. Recall that $x^r_j < x^r_m$. Since I_m is to the left of I_j in $\mathcal{C}_3$, comparing with its input position, I_j must have been moved rightwards in $\mathcal{C}_3$. Thus, I_j is valid. For any interval after m', its position is the same as in $\mathcal{C}_2$, and thus it is valid.

The above proves that $\mathcal{C}_3$ is feasible. In the following, we show that $\mathcal{C}_3$ is an optimal configuration by proving that $\delta(\mathcal{C}_3) \leq \delta(\mathcal{C}_2) = \delta_{opt}$. It is sufficient to show that for any interval j before and including m' in $\mathcal{C}_3$, $d(j, \mathcal{C}_3) \leq \delta_{opt}$.

- Consider any interval j before and including h in $\mathcal{C}_3$. We have $d(j, \mathcal{C}_3) = d(j, \mathcal{C}_{L_1}) \leq \delta(\mathcal{C}_{L_1})$. By lemma condition, $\delta(\mathcal{C}_{L_1}) \leq d(m, \mathcal{C}_{L_2}) \leq \delta(\mathcal{C}_{L_2})$. Since L_2 is consistent with Q_2 and $\mathcal{C}_{L_2}$ is constructed based on the left-possible placement strategy, it holds that $\delta(\mathcal{C}_{L_2}) \leq \delta_{opt}$. Therefore, $d(j, \mathcal{C}_3) \leq \delta_{opt}$.

- Consider interval m'. In the following, we show that $d(m', \mathcal{C}_3) \leq d(m, \mathcal{C}_2)$, which will lead to $d(m', \mathcal{C}_3) \leq \delta_{opt}$ since $d(m, \mathcal{C}_2) \leq \delta_{opt}$.

  By lemma condition, $d(m', \mathcal{C}_{L_1}) \leq \delta(\mathcal{C}_{L_1}) \leq d(m, \mathcal{C}_{L_2})$. As discussed above, $x^r_{m'}(\mathcal{C}_3) = x^r_{m'}(\mathcal{C}_{L_1}) + l$. Therefore, $d(m', \mathcal{C}_3) = d(m', \mathcal{C}_{L_1}) + l$. On the other hand, as discussed above, $x^r_m(\mathcal{C}_2) \geq x^r_m(\mathcal{C}_{L_2}) + l$. Therefore, $d(m, \mathcal{C}_2) \geq d(m, \mathcal{C}_{L_2}) + l$. Due to $d(m', \mathcal{C}_{L_1}) \leq d(m, \mathcal{C}_{L_2})$, we obtain $d(m', \mathcal{C}_3) \leq d(m, \mathcal{C}_2)$.

- Consider any index $j \in S'$. Recall that $m' \leq i < j$ as $S' \subseteq \mathcal{I}[i+1, n]$. Therefore, $x^l_{m'} \leq x^l_j$. On the other hand, $l_{m'}$ is to the right of l_j in $\mathcal{C}_3$. Thus, it holds that $d(j, \mathcal{C}_3) \leq d(m', \mathcal{C}_3)$. We have proved above that $d(m', \mathcal{C}_3) \leq \delta_{opt}$. Hence, we also obtain $d(j, \mathcal{C}_3) \leq \delta_{opt}$.

This proves that $\mathcal{C}_3$ is an optimal configuration. As discussed above, the lemma follows. □

Lemma 6.6.2. *Suppose L_1 and L_2 are two lists of $\mathcal{L}$ whose last indices are the same. Then, if $\delta(\mathcal{C}_{L_1}) \leq \delta(\mathcal{C}_{L_2})$ and $x(\mathcal{C}_{L_1}) \leq x(\mathcal{C}_{L_2})$, then L_1 dominates L_2.*

Proof. Assume L_2 is a canonical list of $\mathcal{I}[1, i]$. Our goal is prove that L_1 is also a canonical list of $\mathcal{I}[1, i]$. To this end, it is sufficient to construct an optimal configuration in which the order the intervals of $\mathcal{I}[1, i]$ is L_1. The proof techniques are similar to (but simpler than) that for Lemma 6.6.1.

Let m be the last index of L_1 and L_2. Let h (resp., g) be the left neighboring index of m in L_1 (resp., L_2).

Since L_2 is a canonical list, there is an optimal list Q that is consistent with L_2. By the definition of g, all indices (if any) strictly between g and m in Q are from $\mathcal{I}[i+1, n]$. Let S denote the set of indices of $\mathcal{I}[i+1, n]$ before g in Q. We consider the general case where $S \neq \emptyset$.

$$Q_1 : \cdots\cdots g, S, m, \cdots\cdots$$
$$Q_2 : \cdots\cdots h, S, m, \cdots\cdots$$

Figure 6.14. Illustrating the two lists Q_1 and Q_2. In Q_1 (resp., Q_2), the indices strictly before S are exactly those in $\mathcal{I}[1, i] \setminus \{m\}$.

As in the proof of Lemma 6.6.1, we can obtain an optimal list Q_1 that is the same as Q except that all indices of S are now right after g in Q_1 (i.e., all indices of Q before g except those in S are still before g in Q_1 with the same relative order, and all indices of Q after g are now after indices of S in Q_1 with the same relative order; e.g., see Fig. 6.14). Therefore, in Q_1, the indices before and including g are exactly those in $\mathcal{I}[1, i] \setminus \{m\}$.

Let Q_2 be another list that is the same as Q_1 except the following (e.g., see Fig. 6.14): We re-arrange the indices before and including g such that they follow exactly the same order as in L_1. Note that L_1 is consistent with Q_2. In the following, we show that Q_2 is an optimal list, which will prove the lemma.

Since Q_1 is an optimal list, there is an optimal configuration $\mathcal{C}_1$ whose interval order is the same as Q_1. Consider the configuration $\mathcal{C}_2$ that is the same as $\mathcal{C}_1$ except the following: For each interval k before and including g, we set the position of I_k the same as its position in $\mathcal{C}_{L_1}$. Hence, the interval order of $\mathcal{C}_2$ is the same as Q_2. In the following, we show that $\mathcal{C}_2$ is an optimal configuration, which will prove that Q_2 is an optimal list.

We first show that $\mathcal{C}_2$ is feasible. For each interval k before and including h, its position in $\mathcal{C}_2$ is the same as that in $\mathcal{C}_{L_1}$, and thus interval k is still valid in $\mathcal{C}_2$. Other intervals are also valid since they do not change their positions from $\mathcal{C}_1$ to $\mathcal{C}_2$. In the following, we show that no two intervals overlap in $\mathcal{C}_2$. Based on our way of constructing $\mathcal{C}_2$, it is sufficient to show that $x_h^r(\mathcal{C}_2) \leq x_t^l(\mathcal{C}_2)$, where t is the right neighboring index of h in Q_2. Note that $x_h^r(\mathcal{C}_2) = x_h^r(\mathcal{C}_{L_1})$ and $x_t^l(\mathcal{C}_2) = x_t^l(\mathcal{C}_1)$. In the following, we prove that $x_h^r(\mathcal{C}_{L_1}) \leq x_t^l(\mathcal{C}_1)$. Depending on whether $x_h^r(\mathcal{C}_{L_1}) \leq x_g^r(\mathcal{C}_{L_2})$, there are two cases.

1. If $x_h^r(\mathcal{C}_{L_1}) \leq x_g^r(\mathcal{C}_{L_2})$, then since L_2 is consistent with Q_1 and $\mathcal{C}_{L_2}$ is constructed based on the left-possible placement strategy, we have $x_g^r(\mathcal{C}_{L_2}) \leq x_g^r(\mathcal{C}_1)$, and thus, $x_h^r(\mathcal{C}_{L_1}) \leq x_g^r(\mathcal{C}_1)$.

On the other hand, note that t is also the right neighboring index of g in Q_1. Since $\mathcal{C}_1$ is feasible, $x^r_g(\mathcal{C}_1) \leq x^l_t(\mathcal{C}_1)$. Thus, we obtain $x^r_h(\mathcal{C}_{L_1}) \leq x^l_t(\mathcal{C}_1)$.

2. Assume $x^r_h(\mathcal{C}_{L_1}) > x^r_g(\mathcal{C}_{L_2})$. By the lemma condition, we have $x^r_m(\mathcal{C}_{L_1}) = x(\mathcal{C}_{L_1}) \leq x(\mathcal{C}_{L_2}) = x^r_m(\mathcal{C}_{L_2})$. Since $x^r_h(\mathcal{C}_{L_1}) > x^r_g(\mathcal{C}_{L_2})$ and both $\mathcal{C}_{L_1}$ and $\mathcal{C}_{L_2}$ are constructed by the left-possible placement strategy, it must be that $x^l_m(\mathcal{C}_{L_1}) = x^l_m(\mathcal{C}_{L_2}) = x^l_m$, i.e., the positions of I_m in both $\mathcal{C}_{L_1}$ and $\mathcal{C}_{L_2}$ are the same as that in the input.

   Since t is in $\mathcal{I}[i+1, n]$ and $m \leq i$, $x^l_m \leq x^l_t$. Since $x^l_t \leq x^l_t(\mathcal{C}_{L_1}) \leq x^l_t(\mathcal{C}_1)$, it holds that $x^l_m \leq x^l_t(\mathcal{C}_1)$. Since I_m is to the right of I_h in the configuration $\mathcal{C}_{L_1}$, $x^r_h(\mathcal{C}_{L_1}) \leq x^l_m(\mathcal{C}_{L_1}) = x^l_m$. Consequently, we obtain $x^r_h(\mathcal{C}_{L_1}) \leq x^l_t(\mathcal{C}_1)$.

This proves that $\mathcal{C}_2$ is feasible. In the sequel we show that $\mathcal{C}_2$ is an optimal configuration by proving that $\delta(\mathcal{C}_2) \leq \delta(\mathcal{C}_1) = \delta_{opt}$. Since the intervals strictly after g do not change their positions from $\mathcal{C}_1$ to $\mathcal{C}_2$, it is sufficient to show that $d(k, \mathcal{C}_2) \leq \delta_{opt}$ for any index k before and including g in $\mathcal{C}_2$.

Since $x^l_k(\mathcal{C}_2) = x^l_k(\mathcal{C}_{L_1})$, $d(k, \mathcal{C}_2) = d(k, \mathcal{C}_{L_1}) \leq \delta(\mathcal{C}_{L_1})$. By lemma condition, $\delta(\mathcal{C}_{L_1}) \leq \delta(\mathcal{C}_{L_2})$. Since L_2 is consistent with Q_1 and $\mathcal{C}_{L_2}$ is constructed based on the left-possible placement strategy, it holds that $\delta(\mathcal{C}_{L_2}) \leq \delta(\mathcal{C}_1) = \delta_{opt}$. Combining the above discussions, we obtain $d(k, \mathcal{C}_2) \leq \delta(\mathcal{C}_{L_1}) \leq \delta(\mathcal{C}_{L_2}) \leq \delta_{opt}$.

This proves that $\mathcal{C}_2$ is an optimal configuration. The lemma thus follows. □

Let $E(\mathcal{L})$ denote the set of last intervals of all lists of $\mathcal{L}$. Our preliminary algorithm guarantees the following property on $E(\mathcal{L})$, which will be useful later for our pruning algorithm given in Section 6.6.2.

Lemma 6.6.3. *$E(\mathcal{L})$ has at most two intervals. Further, if $|E(\mathcal{L})| = 2$, then one interval of $E(\mathcal{L})$ contains the other one in the input.*

Proof. We prove the lemma by induction. Initially, after I_1 is processed, $\mathcal{L}$ consists of the only list $L = \{1\}$. Therefore, $E(\mathcal{L}) = \{1\}$ and the lemma trivially holds.

We assume that the lemma holds after interval I_{i-1} is processed. Let $\mathcal{L}$ be the set after I_i is processed. For differentiation, we let $\mathcal{L}'$ denote the set $\mathcal{L}$ before I_i is processed.

Depending on whether the size of $E(\mathcal{L}')$ is 1 or 2, there are two cases.

The case $|E(\mathcal{L}')| = 1$.. Let m be the only index of $E(\mathcal{L}')$. Hence, for each list $L \in \mathcal{L}'$, m is the last index of L. Depending on whether $x^r_m \leq x^r_i$, there are two subcases.

1. If $x^r_m \leq x^r_i$, then according to our preliminary algorithm, Case I of the algorithm happens on every list $L \in \mathcal{L}'$, and i is appended at the end of L for each $L \in \mathcal{L}'$. Therefore, the last indices of all lists of $\mathcal{L}$ are i, and the lemma statement holds for $E(\mathcal{L})$.

2. If $x^r_m > x^r_i$, then note that $I_i \subseteq I_m$ in the input. Consider any list $L \in \mathcal{L}'$. According to our preliminary algorithm, if $x^l_i \leq x^l_m(\mathcal{C}_L)$, then i is inserted into L right before m; otherwise, i is appended at the end of L, and further, a new list L^* is produced in which m is at the end.

   Therefore, in this case, $E(\mathcal{L})$ has either one index or two indices. If $|E(\mathcal{L})| = 2$, then $E(\mathcal{L}) = \{i, m\}$. Since $I_i \subseteq I_m$ in the input, the lemma statement holds on $E(\mathcal{L})$.

The case $|E(\mathcal{L}')| = 2$.. By induction hypothesis, one interval of $E(\mathcal{L}')$ contains the other one in the input. Let m and m' be the two indices of $E(\mathcal{L}')$, respectively, such that $I_{m'} \subseteq I_m$ in the input. Hence, we have $m < m'$ and $x^r_{m'} \leq x^r_m$.

Depending on the x-coordinates of right endpoints of I_i, I_m, and $I_{m'}$ in the input, there are three subcases: $x^r_m \leq x^r_i$, $x^r_{m'} \leq x^r_i < x^r_m$, and $x^r_i < x^r_{m'}$.

1. If $x^r_m \leq x^r_i$, then for each list $L \in \mathcal{L}'$, Case I of the algorithm happens, and i is appended at the end of L. Therefore, the last indices of all lists of $\mathcal{L}$ are i, and the lemma statement holds for $E(\mathcal{L})$.

2. If $x^r_{m'} \leq x^r_i < x^r_m$, then consider any list $L \in \mathcal{L}'$. If m' is at the end of L, then Case I happens and i is appended at the end of L. If m is at the end of L, then either Case II or Case III of the algorithm happens. Hence, either i or m will be the last index of L; if a new list L^* is produced in Case III, then its last index is m.

Therefore, after every list of $\mathcal{L}'$ is processed, the last index of each list of $\mathcal{L}$ is either m or i, i.e., $E(\mathcal{L}) = \{m, i\}$. Note that I_i is contained in I_m in the input. Hence, the lemma statement holds for $E(\mathcal{L})$.

3. If $x_i^r < x_{m'}^r$, then I_i is contained in both I_m and $I_{m'}$ in the input. Consider any list $L \in \mathcal{L}'$. Regardless of whether the last index is m or m', Case I does not happen.

   We claim that Case III does not happen either. We prove the claim only for the case where the last index of L is m (the other case can be proved similarly). Indeed, in the configuration $\mathcal{C}_L$, it holds that $x_{m'}^r \leq x_{m'}^r(\mathcal{C}_L)$. Since m is the last index of L, we have $x_{m'}^r(\mathcal{C}_L) \leq x_m^l(\mathcal{C}_L)$. Since $x_i^r < x_{m'}^r$, we obtain $x_i^l \leq x_i^r < x_{m'}^r \leq x_{m'}^r(\mathcal{C}_L) \leq x_m^l(\mathcal{C}_L)$. This implies that Case III of the algorithm cannot happen.

   Hence, Case II happens, and i is inserted into L right before the last index. Therefore, the last indices of all lists of $\mathcal{L}$ are either m or m'. The lemma statement holds for $E(\mathcal{L})$.

This proves the lemma. □

6.6.2 A Pruning Procedure

Based on Lemmas 6.6.1 and 6.6.2, we present an algorithm that prunes redundant lists from $\mathcal{L}$ after each step for processing an interval I_i. In the following, we describe the algorithm, whose implementation is discussed in Section 6.6.3.

By Lemma 6.6.3, $E(\mathcal{L})$ has at most two indices. If $E(\mathcal{L})$ has two indices, we let m and m' denote the two indices, respectively, such that $I_{m'} \subseteq I_m$ in the input. If $E(\mathcal{L})$ has only one index, let m denote it and m' is undefined. Let $\mathcal{L}_1$ (resp., $\mathcal{L}_2$) denote the set of lists of $\mathcal{L}$ whose last indices are m' (resp., m), and $\mathcal{L}_1 = \emptyset$ if and only if m' is undefined.

Our algorithm maintains several invariants regarding certain monotonicity properties, as follows, which are crucial to our efficient implementation.

1. $\mathcal{L}$ contains a canonical list of $\mathcal{I}[1, i]$.

2. For any two lists L_1 and L_2 of $\mathcal{L}$, $x(\mathcal{C}_{L_1}) \neq x(\mathcal{C}_{L_2})$ and $\delta(\mathcal{C}_{L_1}) \neq \delta(\mathcal{C}_{L_2})$.

3. If $\mathcal{L}_1 \neq \emptyset$, then for any lists $L_1 \in \mathcal{L}_1$ and $L_2 \in \mathcal{L}_2$, $x(\mathcal{C}_{L_1}) < x(\mathcal{C}_{L_2})$.

4. For any two lists L_1 and L_2 of $\mathcal{L}$, $x(\mathcal{C}_{L_1}) < x(\mathcal{C}_{L_2})$ if and only if $\delta(\mathcal{C}_{L_1}) > \delta(\mathcal{C}_{L_2})$. In other words, if we order the lists L of $\mathcal{L}$ increasingly by the values $x(\mathcal{C}_L)$, then the values $\delta(\mathcal{C}_L)$ are sorted decreasingly.

After I_n is processed, by the algorithm invariants, if L is the list of $\mathcal{L}$ with minimum $\delta(\mathcal{C}_L)$, then L is an optimal list and $\delta_{opt} = \delta(\mathcal{C}_L)$.

Initially after the first interval I_1 is processed, $\mathcal{L}$ has only one list $L = \{1\}$, and thus, all algorithm invariants trivially hold. In general, suppose the first $i-1$ intervals have been processed and all algorithm invariants hold on $\mathcal{L}$. In the following, we discuss the general step for processing interval I_i.

For differentiation, we let $\mathcal{L}'$ refer to the original set $\mathcal{L}$ before interval i is processed. Similarly, we use $\mathcal{L}'_1$ and $\mathcal{L}'_2$ to refer to $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. Let $L'_1, L'_2, \ldots, L'_a$ be the lists of $\mathcal{L}'$ sorted with $x(\mathcal{C}_{L'_1}) < x(\mathcal{C}_{L'_2}) < \cdots < x(\mathcal{C}_{L'_a})$, where $a = |\mathcal{L}'|$. By the fourth invariant, we have $\delta(\mathcal{C}_{L'_1}) > \delta(\mathcal{C}_{L'_2}) > \cdots > \delta(\mathcal{C}_{L'_a})$. If $\mathcal{L}'_1 = \emptyset$, let $b = 0$; otherwise, let b be the largest index such that $L'_b \in \mathcal{L}'_1$, and by the third algorithm invariant, $\mathcal{L}'_1 = \{L'_1, \ldots, L'_b\}$ and $\mathcal{L}'_2 = \{L'_{b+1}, \ldots, L_a\}$. Depending on whether $\mathcal{L}'_1 = \emptyset$, there are two main cases.

The Case $\mathcal{L}'_1 = \emptyset$

In this case, for each list $L' \in \mathcal{L}'$, its last index is m. Depending on whether $x^r_m \leq x^r_i$, there are two subcases.

The first subcase $x^r_m \leq x^r_i$.. In this case, according to the preliminary algorithm, for each list $L'_j \in \mathcal{L}'$, Case I happens and i is appended at the end of L'_j, and we use L_j to refer to the updated list of L'_j with i. According to our left-possible placement strategy, $x^l_i(\mathcal{C}_{L_j}) = \max\{x(\mathcal{C}_{L'_j}), x^l_i\}$. Thus, $x(\mathcal{C}_{L_j}) = x^l_i(\mathcal{C}_{L_j}) + |I_i|$ and $d(i, \mathcal{C}_{L_j}) = x^l_i(\mathcal{C}_{L_j}) - x^l_i$.

As the index j increases from 1 to a, since the value $x(\mathcal{C}_{L'_j})$ strictly increases, $x^l_i(\mathcal{C}_{L_j})$ (and thus $x(\mathcal{C}_{L_j})$ and $d(i, \mathcal{C}_{L_j})$) is monotonically increasing (it may first be constant and then strictly increases after some index, say, a_1). Formally, we define a_1 as follows. If $x(\mathcal{C}_{L'_1}) > x^l_i$, then let $a_1 = 0$; otherwise, define a_1 to be the largest index $j \in [1, a]$ such that $x(\mathcal{C}_{L'_j}) \leq x^l_i$ (e.g., see Fig. 6.15). In the following, we first assume $a_1 \neq 0$. As

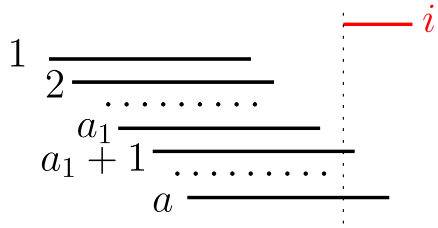


Figure 6.15. Illustrating the definition of a_1. The black segments show the positions of interval m in the configurations $\mathcal{C}_{L'_j}$ for $j \in [1, a]$, and the numbers on the left side are the indices of the lists. The red segment shows the interval i in the input position.

discussed above, as j increases in $[1, a]$, $x^l_i(\mathcal{C}_{L_j})$ is constant on $j \in [1, a_1]$ and strictly increases on $j \in [a_1, a]$.

Now consider the value $\delta(\mathcal{C}_{L_j})$, which is equal to $\max\{\delta(\mathcal{C}_{L'_j}), d(i, \mathcal{C}_{L_j})\}$ by Observation 6.4.1. Recall that $\delta(\mathcal{C}_{L'_j})$ is strictly decreasing on $j \in [1, a]$. Observe that $d(i, \mathcal{C}_{L_j})$ is 0 on $j \in [1, a_1]$ and strictly increases on $j \in [a_1, a]$. This implies that $\delta(\mathcal{C}_{L_j})$ on $j \in [1, a]$ is a unimodal function, i.e., it first strictly decreases and then strictly increases after some index, say, a_2. Formally, let a_2 be the largest index $j \in [a_1 + 1, a]$ such that $\delta(\mathcal{C}_{L_{j-1}}) > \delta(\mathcal{C}_{L_j})$, and if no such index j exists, then let $a_2 = a_1$. The following lemma is proved based on Lemma 6.6.2.

Lemma 6.6.4. *1. If $a_1 > 1$, then for each $j \in [1, a_1 - 1]$, L_{a_1} dominates L_j.*

2. If $a_2 < a$, then for each $j \in [a_2 + 1, a]$, L_{a_2} dominates L_j.

Proof. 1. Let $k = a_1$ and assume $k > 1$. Consider any $j \in [1, k-1]$. By the definition of a_1, $x^l_i(\mathcal{C}_{L_j}) = x^l_i(\mathcal{C}_{L_k}) = x^l_i$. Therefore, $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L_k}) = x^l_i + |I_i|$. Since $d(i, \mathcal{C}_{L_j}) = d(i, \mathcal{C}_{L_k}) = 0$, we have $\delta(\mathcal{C}_{L_j}) = \delta(\mathcal{C}_{L'_j})$ and $\delta(\mathcal{C}_{L_k}) = \delta(\mathcal{C}_{L'_k})$. Since $j < k$, $\delta(\mathcal{C}_{L'_j}) > \delta(\mathcal{C}_{L'_k})$. Thus, we obtain $\delta(\mathcal{C}_{L_j}) > \delta(\mathcal{C}_{L_k})$.

Since $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L_k})$, $\delta(\mathcal{C}_{L_j}) > \delta(\mathcal{C}_{L_k})$, and the last indices of L_j and L_k are both i, by Lemma 6.6.2, L_k dominates L_j.

2. Let $k = a_2$ and assume $k < a$. Consider any $j \in [k+1, a]$. As discussed before, $x(\mathcal{C}_{L_j})$ is monotonically increasing on $j \in [1, a]$. Thus, $x(\mathcal{C}_{L_k}) \leq x(\mathcal{C}_{L_j})$. By the definition of a_2 and since $\delta(\mathcal{C}_{L_j})$ is a unimodal function on $j \in [1, a]$, it holds that $\delta(\mathcal{C}_{L_k}) \leq \delta(\mathcal{C}_{L_j})$. By Lemma 6.6.2, L_k dominates L_j.

This proves the lemma. ∎

By Lemma 6.6.4, we let $\mathcal{L} = \{L_j \mid a_1 \leq j \leq a_2\}$. The above is for the general case where $a_1 \neq 0$. If $a_1 = 0$, then we let $\mathcal{L} = \{L_j \mid 1 \leq j \leq a_2\}$.

Observation 6.6.5. *All algorithm invariants hold for $\mathcal{L}$.*

Proof. By Lemma 6.6.4, the lists that have been removed are redundant. Hence, $\mathcal{L}$ contains a canonical list of $\mathcal{I}[1, i]$ and the first algorithm invariant holds.

By our definitions of a_1 and a_2, when j increases in $[a_1, a_2]$, $x(\mathcal{C}_{L_j})$ strictly increases and $\delta(\mathcal{C}_{L_j})$ strictly decreases. Therefore, the last three algorithm invariants hold. $\square$

The following lemma will be quite useful for the algorithm implementation given later in Section 6.6.3.

Lemma 6.6.6. *If $a_1 < a_2$, then for each $j \in [a_1 + 1, a_2]$, $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$. For each list $L_j \in \mathcal{L}$ with $j \neq a_2$, $\delta(\mathcal{C}_{L_j}) = \delta(\mathcal{C}_{L'_j})$.*

Proof. By the definition of a_1, for any $j \in [a_1 + 1, a]$, it always holds that $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$. This proves the first lemma statement.

Recall that $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(i, \mathcal{C}_{L_j})\}$ for each $j \in [1, a]$.

Consider any list L_j with $j \neq a_2$. Assume to the contrary that $\delta(\mathcal{C}_{L_j}) \neq \delta(\mathcal{C}_{L'_j})$. Then, $\delta(\mathcal{C}_{L_j}) = d(i, \mathcal{C}_{L_j})$. Since $\delta(\mathcal{C}_{L_j}) = d(i, \mathcal{C}_{L_j}) < d(i, \mathcal{C}_{L_{a_2}})$, we obtain $\delta(\mathcal{C}_{L_j}) \leq \delta(\mathcal{C}_{L_{a_2}})$, which contradicts with $\delta(\mathcal{C}_{L_j}) > \delta(\mathcal{C}_{L_{a_2}})$. $\square$

The second subcase $x^r_m > x^r_i$.. In this case, for each list $L'_j \in \mathcal{L}'$, according to our preliminary algorithm, depending on whether $x^l_i \leq x^l_m(\mathcal{C}_{L'_j})$, either Case II or Case III can happen. If $x^l_i \leq x^l_m(\mathcal{C}_{L'_1})$, then let $c = 0$; otherwise, let c be the largest index j such that $x^l_i > x^l_m(\mathcal{C}_{L'_j})$ (e.g., see Fig. 6.16). In the following, we first consider the general case where $1 \leq c < a$.

For each $j \in [1, c]$, observe that $x^l_m(\mathcal{C}_{L'_j}) = x(\mathcal{C}_{L'_j}) - |I_m| \leq x(\mathcal{C}_{L'_c}) - |I_m| = x^l_m(\mathcal{C}_{L'_c}) < x^l_i$. According to our preliminary algorithm, Case III happens, and thus L'_j will produce two lists: the list L_j by appending i at the end of L'_j, and the new list L^*_j by inserting i in front of m in L'_j. Further, according to our left-possible placement strategy, $x^l_i(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j})$ in $\mathcal{C}_{L_j}$, and $x^l_i(\mathcal{C}_{L^*_j}) = x^l_i$ and $x^l_m(\mathcal{C}_{L^*_j}) = x^r_i$ in $\mathcal{C}_{L^*_j}$. By Observation 6.4.5, $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(i, \mathcal{C}_{L_j})\}$ and $\delta(\mathcal{C}_{L^*_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L^*_j})\}$.

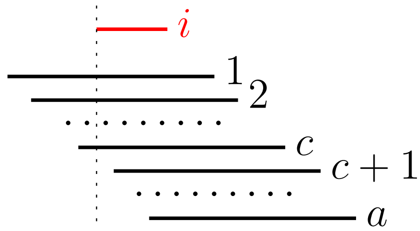


Figure 6.16. Illustrating the definition of c. The black segments show the positions of interval m in the configurations $\mathcal{C}_{L'_j}$ for $j \in [1, a]$, and the numbers on the right side are the indices of the lists. The red segment shows the interval i in the input position.

Observation 6.6.7. *$\delta(\mathcal{C}_{L^*_c}) \leq \delta(\mathcal{C}_{L^*_j})$ for any $j \in [1, c]$.*

Proof. For any $j \in [1, c]$, note that $d(m, \mathcal{C}_{L^*_j}) = x^l_m(\mathcal{C}_{L^*_j}) - x^l_m = x^r_i - x^l_m$. Therefore, $d(m, \mathcal{C}_{L^*_j})$ is the same for all $j \in [1, c]$. On the other hand, we have $\delta(\mathcal{C}_{L'_j}) \geq \delta(\mathcal{C}_{L'_c})$. Thus, $\delta(\mathcal{C}_{L^*_c}) \leq \delta(\mathcal{C}_{L^*_j})$. □

By the above observation and Lemma 6.4.7, among the new lists L^*_j with $j = 1, 2, \ldots, c$, only L^*_c needs to be kept.

For each $j \in [1, c]$, note that $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$. Since $x(\mathcal{C}_{L'_j})$ is strictly increasing on $j \in [1, c]$, $x(\mathcal{C}_{L_j})$ is also strictly increasing on $j \in [1, c]$. Since $d(i, \mathcal{C}_{L_j}) = x^l_i(\mathcal{C}_{L_j}) - x^l_i = x(\mathcal{C}_{L'_j}) - x^l_i$ for any $j \in [1, c]$, $d(i, \mathcal{C}_{L_j})$ also strictly increases on $j \in [1, c]$. Further, since $\delta(\mathcal{C}_{L'_j})$ strictly decreases on $j \in [1, c]$, $\delta(\mathcal{C}_{L_j})$, which is equal to $\max\{\delta(\mathcal{C}_{L'_j}), d(i, \mathcal{C}_{L_j})\}$, is a unimodal function (i.e., it first strictly decreases and then strictly increases). Let c_1 be the smallest index $j \in [1, c-1]$ such that $\delta(\mathcal{C}_{L_j}) \leq \delta(\mathcal{C}_{L_{j+1}})$, and if such an index j does not exist, then let $c_1 = c$.

Lemma 6.6.8. *If $c_1 < c$, then L_{c_1} dominates L_j for any $j \in [c_1 + 1, c]$.*

Proof. Consider any $j \in [c_1 + 1, c]$. Since $\delta(\mathcal{C}_{L_j})$ is a unimodal function on $j \in [1, c]$, by the definition of c_1, $\delta(\mathcal{C}_{L_{c_1}}) \leq \delta(\mathcal{C}_{L_j})$. Recall that $x(\mathcal{C}_{L_{c_1}}) \leq x(\mathcal{C}_{L_j})$. Since the last indices of L_{c_1} and L_j are both i, by Lemma 6.6.2, L_{c_1} dominates L_j. □

By the preceding lemma, if $c_1 < c$, then we do not have to keep the lists $L_{c_1+1}, \ldots, L_c$ in $\mathcal{L}$. Let $S_1 = \{L_1, \ldots, L_{c_1}\}$.

Consider any index $j \in [c + 1, a]$. By the definition of c and also due to that $x(\mathcal{C}_{L'_k})$ is strictly increasing on $k \in [1, a]$, it holds that $x^l_m(\mathcal{C}_{L'_j}) \geq x^l_i$, and thus Case II of the preliminary algorithm happens on L'_j and L_j is obtained by inserting i right before m in L'_j. By Observation 6.4.3, $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L_j})\}$. Note that $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$ and $x^r_m(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L_j})$. As j increases in $[c+1, a]$, since $x(\mathcal{C}_{L'_j})$ strictly increases, both $x(\mathcal{C}_{L_j})$ and $d(m, \mathcal{C}_{L_j})$ strictly increase. Since $\delta(\mathcal{C}_{L'_j})$ is strictly decreasing on $j \in [c+1, a]$, we obtain that $\delta(\mathcal{C}_{L_j})$ is a unimodal function on $j \in [c+1, a]$ (i.e., it first strictly decreases and then strictly increases).

Let $S = \{L_1, \ldots, L_c, L^*_c, L_{c+1}, \ldots, L_a\}$. For convenience, we use $L_{c+0.5}$ to refer to L^*_c (and $L'_{c+0.5}$ refers to L'_c); in this way, the indices of the ordered lists of S are sorted. Consider the subsequence of the lists of S from $L_{c+0.5}$ to the end (including $L_{c+0.5}$). Define c_2 to be the index of the first list L_j such that $\delta(\mathcal{C}_{L_j}) \leq \delta(\mathcal{C}_L)$, where L is the right neighboring list of L_j in S; if such a list L_j does not exist, then we let $c_2 = a$.

Observation 6.6.9. *As j increases in $[1, a]$, $x(\mathcal{C}_{L_j})$ is strictly increasing except that $x(\mathcal{C}_{L_{c+0.5}}) = x(\mathcal{C}_{L_{c+1}})$ may be possible.*

Proof. Recall that $x(\mathcal{C}_{L_j})$ is strictly increasing on $j \in [1, c]$ and $j \in [c+1, a]$, respectively. Let $l = |I_i| + |I_m|$. Note that $x(\mathcal{C}_{L_c}) = x^l_m(\mathcal{C}_{L'_c}) + l$, $x(\mathcal{C}_{L^*_c}) = x^l_i + l$, and $x(\mathcal{C}_{L_{c+1}}) = x^l_m(\mathcal{C}_{L'_{c+1}}) + l$. By our definition of c, $x^l_m(\mathcal{C}_{L'_c}) < x^l_i \leq x^l_m(\mathcal{C}_{L'_{c+1}})$. Thus, $x(\mathcal{C}_{L_c}) < x(\mathcal{C}_{L^*_c}) \leq x(\mathcal{C}_{L_{c+1}})$. This shows that $x(\mathcal{C}_{L_j})$ is strictly increasing on $j \in [1, a]$ except that $x(\mathcal{C}_{L^*_c}) = x(\mathcal{C}_{L_{c+1}})$ may be possible. ☐

Lemma 6.6.10. *1. If $c_2 < a$, then L_{c_2} dominates L_j for any $L_j \in S$ with $j > c_2$.*

2. If $c_2 \geq c+1$ and $x(\mathcal{C}_{L_{c+0.5}}) = x(\mathcal{C}_{L_{c+1}})$, then L_{c+1} dominates $L_{c+0.5}$.

Proof. We first show that $\delta(\mathcal{C}_{L_j})$ is a unimodal function on $j \in [c+0.5, a]$.

Recall that for each $j \in [c+1, a]$, $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L_j})\}$, and $\delta(\mathcal{C}_{L^*_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L^*_j})\}$. For each $j \in [c+0.5, a]$, since m is the last index of L_j, we have $d(m, \mathcal{C}_{L_j}) = x(\mathcal{C}_{L_j}) - x^r_m$. By Observation 6.6.9, $d(m, \mathcal{C}_{L_j})$ is strictly increasing on $[c+0.5, a]$ except that $d(m, \mathcal{C}_{L_{c+0.5}}) = d(m, \mathcal{C}_{L_{c+1}})$ may be possible. Since $\delta(\mathcal{C}_{L'_j})$ on $j \in [1, a]$ is strictly decreasing, $\delta(\mathcal{C}_{L_j})$ is a unimodal function on $j \in [c+0.5, a]$.

By the definition of c_2, $\delta(\mathcal{C}_{L_j})$ is strictly decreasing on $[c+0.5, c_2]$ and monotonically increasing on $[c_2, a]$.

Consider any list $L_j \in S$ with $j > c_2$. By our previous discussion, $\delta(\mathcal{C}_{L_{c_2}}) \leq \delta(\mathcal{C}_{L_j})$ and $x(\mathcal{C}_{L_{c_2}}) \leq x(\mathcal{C}_{L_j})$. Since the last indices of both L_{c_2} and L_j are m, by Lemma 6.6.2, L_{c_2} dominates L_j.

If $c_2 \geq c+1$ and $x(\mathcal{C}_{L_{c+0.5}}) = x(\mathcal{C}_{L_{c+1}})$, by the definition of c_2, $\delta(\mathcal{C}_{L_{c+0.5}}) > \delta(\mathcal{C}_{L_{c+1}})$. Since the last indices of both $L_{c+0.5}$ and L_{c+1} are m, by Lemma 6.6.2, L_{c+1} dominates $L_{c+0.5}$. The lemma thus follows. □

Let $S_2 = \{L_{c+0.5}, L_{c+1}, \ldots, L_{c_2}\}$ and we remove $L_{c+0.5}$ from S_2 if $c_2 \geq c+1$ and $x(\mathcal{C}_{L_{c+0.5}}) = x(\mathcal{C}_{L_{c+1}})$. In the following, we combine S_1 and S_2 to obtain the set $\mathcal{L}$. We consider the lists of S_2 in order. Define c' to be the index j of the first list L_j such that $\delta(\mathcal{C}_{L_{c_1}}) > \delta(\mathcal{C}_{L_j})$, and if no such list L_j exists, then let $c' = c_2 + 1$.

Lemma 6.6.11. *If $L_{c'}$ is not the first list of S_2 or $c' = c_2 + 1$, then for each list L_j of S_2 with $j < c'$, L_{c_1} dominates L_j.*

Proof. We assume that $L_{c'}$ is not the first list of S_2 or $c' = c_2 + 1$.

Note that we have proved in the proof of Lemma 6.6.10 that $\delta(\mathcal{C}_{L_j})$ on $j \in [c+0.5, c_2]$ is strictly decreasing. By the definition of c', it holds that $\delta(\mathcal{C}_{L_{c_1}}) \leq \delta(\mathcal{C}_{L_j})$ for any $L_j \in S_2$ with $j < c'$.

Consider any list L_j of S_2 with $j < c'$.

Recall that $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L_j})\}$. We claim that $\delta(\mathcal{C}_{L_j}) = d(m, \mathcal{C}_{L_j})$. Indeed, note that $\delta(\mathcal{C}_{L'_j}) \leq \delta(\mathcal{C}_{L'_{c_1}}) \leq \delta(\mathcal{C}_{L_{c_1}})$. Since $\delta(\mathcal{C}_{L_{c_1}}) \leq \delta(\mathcal{C}_{L_j})$, we obtain $\delta(\mathcal{C}_{L'_j}) \leq \delta(\mathcal{C}_{L_j})$, and thus, $\delta(\mathcal{C}_{L_j}) = d(m, \mathcal{C}_{L_j})$.

Consequently, we have $\delta(\mathcal{C}_{L_{c_1}}) \leq d(m, \mathcal{C}_{L_j})$ and $x(\mathcal{C}_{L_{c_1}}) \leq x(\mathcal{C}_{L_j})$ (by Observation 6.6.9). Further, the last index of L_{c_1} is i and the last index of L_j is m, with $x_i^r \leq x_m^r$. By Lemma 6.6.1, L_{c_1} dominates L_j.

The lemma thus follows. □

We remove from S_2 all lists L_j with $j < c'$, and let $\mathcal{L} = S_1 \cup S_2$. In general, if $c' \neq c_2 + 1$, then $\mathcal{L} = \{L_1, \ldots, L_{c_1}, L_{c'}, \ldots, L_{c_2}\}$; otherwise, $\mathcal{L} = \{L_1, \ldots, L_{c_1}\}$.

The above discussion is for the general case where $1 \leq c < a$. If $c = 0$, then L_c^*, c_1 and c' are all undefined, and we have $\mathcal{L} = \{L_1, \ldots, L_{c_2}\}$. If $c = a$, then $\mathcal{L} = \{L_1, \ldots, L_{c_1}\}$ if $\delta(L_{c_1}) \leq \delta(L_c^*)$ and $\mathcal{L} = \{L_1, \ldots, L_{c_1}, L_c^*\}$ otherwise.

Observation 6.6.12. *All algorithm invariants hold on $\mathcal{L}$.*

Proof. We only consider the most general case where $1 \leq c < a$ and $c' \neq c_2 + 1$, since other cases can be proved in a similar but easier way.

By Lemmas 6.6.8, 6.6.10, and 6.6.11, all pruned lists are redundant and thus $\mathcal{L}$ contains a canonical list of $\mathcal{I}[1, i]$. The first algorithm invariant holds.

If $x(\mathcal{C}_{L_{c+0.5}}) = x(\mathcal{C}_{L_{c+1}})$, then $L_{c+0.5}$ and L_{c+1} cannot be both in $\mathcal{L}$ by Lemma 6.6.10(2). Thus, by Observation 6.6.9, $x(\mathcal{C}_{L_j})$ strictly increases in $[1, a]$. Recall that for any list $L_j \in \mathcal{L}$, the last index of L_j is i if $j \leq c_1$ and m otherwise. Recall that I_i is contained in I_m in the input. Thus, the fourth algorithm invariant holds.

Further, our definitions of c_1, c', and c_2 guarantee that $\delta(\mathcal{C}_L)$ on all lists L following their order in $\mathcal{L}$ is strictly decreasing. Therefore, the other two algorithm invariants also hold. □

The following lemma will be useful for the algorithm implementation.

Lemma 6.6.13. *For each list $L_j \in \mathcal{L}$, if $L_j \neq L_c^*$, then $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$; if $L_j \notin \{L_c^*, L_{c_1}, L_{c_2}\}$, then $\delta(\mathcal{C}_{L_j}) = \delta(\mathcal{C}_{L'_j})$.*

Proof. If $L_j \neq L_c^*$, then we have discussed before that $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$ always holds regardless of whether the last index of L_j is i or m.

If $L_j \notin \{L_c^*, L_{c_1}, L_{c_2}\}$, assume to the contrary that $\delta(\mathcal{C}_{L_j}) \neq \delta(\mathcal{C}_{L'_j})$. Then, since $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(k, \mathcal{C}_{L_j})\}$, we obtain that $\delta(\mathcal{C}_{L_j}) = d(k, \mathcal{C}_{L_j})$, where k is the last index of $\mathcal{C}_{L_j}$ (k is i if $j \leq c$ and m otherwise). Note that j is either in $[1, c_1]$ or $[c', c_2]$. We discuss the two cases below.

1. If $j \in [1, c_1]$, then the last index of L_j is i. Since $L_j \neq L_{c_1}$, $j < c_1$ holds. We have discussed before that $d(i, \mathcal{C}_{L_j}) \leq d(i, \mathcal{C}_{L_{c_1}})$. Thus, we can deduce $\delta(\mathcal{C}_{L_j}) = d(i, \mathcal{C}_{L_j}) \leq d(i, \mathcal{C}_{L_{c_1}}) \leq \delta(\mathcal{C}_{L_{c_1}})$. However, we have already proved that $\delta(\mathcal{C}_{L_j}) > \delta(\mathcal{C}_{L_{c_1}})$. Thus, we obtain contradiction.

2. If $j \in [c', c_2]$, the analysis is similar. In this case the last index of L_j is m and $j < c_2$. Since $j < c_2$, we have discussed before that $d(m, \mathcal{C}_{L_j}) \leq d(m, \mathcal{C}_{L_{c_2}})$. Thus, we can deduce $\delta(\mathcal{C}_{L_j}) = d(m, \mathcal{C}_{L_j}) \leq d(m, \mathcal{C}_{L_{c_2}}) \leq \delta(\mathcal{C}_{L_{c_2}})$. However, we have already proved that $\delta(\mathcal{C}_{L_j}) > \delta(\mathcal{C}_{L_{c_2}})$. Thus, we obtain contradiction.

The lemma thus follows. □

The Case $\mathcal{L}'_1 \neq \emptyset$

We then consider the case where $\mathcal{L}'_1 \neq \emptyset$. In this case, recall that $\mathcal{L}'_1 = \{L'_1, \ldots, L'_b\}$ and $\mathcal{L}'_2 = \{L'_{b+1}, \ldots, L'_a\}$. For each $L'_j \in \mathcal{L}'$, the last index of L'_j is m' if $j \leq b$ and m otherwise. Recall that $I_{m'} \subseteq I_m$ in the input. As in the proof of Lemma 6.6.3, there are three subcases: $x^r_i \geq x^r_m$, $x^r_{m'} \leq x^r_i < x^r_m$, and $x^r_i < x^r_{m'}$.

The first subcase $x^r_i \geq x^r_m$.. In this case, for each $L'_j \in \mathcal{L}'$, Case I of the preliminary algorithm happens and L_j is obtained by appending i at the end of L'_j. Our pruning procedure for this subcase is similar to the first subcase in Section 6.6.2, and we briefly discuss it below.

First, for each $L'_j \in \mathcal{L}'$, $x^l_i(\mathcal{C}_{L_j}) = \max\{x(\mathcal{C}_{L'_j}), x^l_i\}$ and $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(i, \mathcal{C}_{L_j})\}$. We define a_1 and a_2 in exactly the same way as in the first subcase of Section 6.6.2, and further, Lemma 6.6.4 still holds. Similarly, we let $\mathcal{L}$ consist of only those lists L_j with $j \in [a_1, a_2]$. By the similar analysis, Observation 6.6.5 and Lemma 6.6.6 still hold. We omit the details.

The second subcase $x^r_{m'} \leq x^r_i < x^r_m$.. In this case, we first apply the similar pruning procedure for the first (resp., second) subcase of Section 6.6.2 to set $\mathcal{L}'_1$ (resp., $\mathcal{L}'_2$), and then we combine the results. The details are given below.

For set $\mathcal{L}'_1$, the last indices of all lists of $\mathcal{L}'_1$ are m'. Since $x^r_{m'} \leq x^r_i$, for each $L'_j \in \mathcal{L}'_1$, Case I of the preliminary algorithm happens and L_j is obtained by appending i at the end of L'_j. We define a_1 and a_2 in the similar way as in the first subcase of Section 6.6.2 but with respect to the indices in $[1, b]$. In fact, since $x^r_i < x^r_m$, it holds that $x^l_i \leq x^r_i \leq x^r_m \leq x(\mathcal{C}_{L'_1})$, and consequently, $a_1 = 0$. Similarly, Lemma 6.6.4 also holds with respect to the indices of $[1, b]$. Further, as j increases in $[1, a_2]$, $x(\mathcal{C}_{L_j})$ is strictly increasing and $\delta(\mathcal{C}_{L_j})$ is strictly decreasing. Let $S'_1 = \{L_1, L_2, \ldots, L_{a_2}\}$.

For set $\mathcal{L}'_2$, the last indices of all its lists are m. Since $x^r_i < x^r_m$, for each list $L'_j \in \mathcal{L}_2$, either Case II or Case III of the algorithm happens. We define c in the similar way as in the second subcase of Section 6.6.2 but with respect to the indices of $[b+1, a]$. Specifically, if $x^l_i \leq x^l_m(\mathcal{C}_{L'_{b+1}})$, then let $c = b$; otherwise, let c be the largest index $j \in [b+1, a]$ such that $x_i^l > x_m^l(\mathcal{C}_{L'_j})$. We consider the most general case where $b+1 \leq c < a$ (other cases are similar but easier).

For each $j \in [b+1, c]$, there is also a new list L_j^*. Similar to Observation 6.6.7, $\delta(\mathcal{C}_{L_c^*}) \leq \delta(\mathcal{C}_{L_j^*})$ for any $j \in [b+1, c]$. Hence, among the new lists L_j^* with $j = b+1, \ldots, c$, only L_c^* needs to be kept. Let $S' = \{L_{b+1}, \ldots, L_c, L_c^*, L_{c+1}, \ldots, L_a\}$. We also use $L_{c+0.5}$ to refer to L_c^*. We define the three indices c_1, c_2, and c' in the similar way as in the second subcase of Section 6.6.2 but with respect to the ordered lists in S'. Similarly, Observation 6.6.9, Lemmas 6.6.8, 6.6.10, and 6.6.11 all hold with respect to the lists in S'. Let $S'_2 = \{L_{b+1}, \ldots, L_{c_1}, L_{c'}, \ldots, L_{c_2}\}$.

Finally, we combine the lists of the two sets S'_1 and S'_2 to obtain $\mathcal{L}$, as follows. Recall that L_{a_2} is the last list of S'_1. We consider the lists of S'_2 in order. Define b' to be the index j of the first list L_j of S'_2 such that $\delta(\mathcal{C}_{L_{a_2}}) > \delta(\mathcal{C}_{L_j})$, and if no such list L_j exists, then let $b' = c_2 + 1$.

Lemma 6.6.14. *1. $x(\mathcal{C}_{L_{a_2}}) < x(\mathcal{C}_{L_{b+1}})$.*

2. If $b' > b+1$, then L_{a_2} dominates L_j for any list $L_j \in S'_2$ with $j < b'$.

Proof. For L_{a_2}, since $a_1 = 0$, we have $x(\mathcal{C}_{L_{a_2}}) = x(\mathcal{C}_{L'_{a_2}}) + |I_i|$. For L_{b+1}, it holds that $x(\mathcal{C}_{L_{b+1}}) = x(\mathcal{C}_{L'_{b+1}}) + |I_i|$. Since $x(\mathcal{C}_{L'_{a_2}}) < x(\mathcal{C}_{L'_{b+1}})$, we have $x(\mathcal{C}_{L_{a_2}}) < x(\mathcal{C}_{L_{b+1}})$. This proves the first statement of the lemma.

Next we prove the second lemma statement. Assume $b' > b+1$. Consider any list $L_j \in S'_2$ with $j < b'$. In the following, we show that L_{a_2} dominates L_j.

Recall that the values $\delta(L)$ of the lists L of S'_2 are strictly decreasing following their order in S'_2. By the definition of b', $\delta(\mathcal{C}_{L_{a_2}}) \leq \delta(\mathcal{C}_{L_j})$. Note that the last index of L_j can be either i or m, and the last index of L_{a_2} is i.

If the last index of L_j is i, then since $\delta(\mathcal{C}_{L_{a_2}}) \leq \delta(\mathcal{C}_{L_j})$ and $x(\mathcal{C}_{L_{a_2}}) < x(\mathcal{C}_{L_{b+1}}) \leq x(\mathcal{C}_{L_j})$, by Lemma 6.6.2, L_{a_2} dominates L_j.

If the last index of L_j is m, then $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L_j})\}$. Recall that $\delta(\mathcal{C}_{L_{a_2}}) = \max\{\delta(\mathcal{C}_{L'_{a_2}}), d(i, \mathcal{C}_{L_{a_2}})\}$ and $\delta(\mathcal{C}_{L'_{a_2}}) > \delta(\mathcal{C}_{L'_j})$. Due to $\delta(\mathcal{C}_{L_{a_2}}) \leq \delta(\mathcal{C}_{L_j})$, we can deduce $\delta(\mathcal{C}_{L'_j}) < \delta(\mathcal{C}_{L'_{a_2}}) \leq \delta(\mathcal{C}_{L_{a_2}}) \leq \delta(\mathcal{C}_{L_j})$. Therefore, $\delta(\mathcal{C}_{L_{a_2}}) \leq \delta(\mathcal{C}_{L_j}) = d(m, \mathcal{C}_{L_j})$. Again, $x(\mathcal{C}_{L_{a_2}}) < x(\mathcal{C}_{L_{b+1}}) \leq x(\mathcal{C}_{L_j})$. Since the last index of L_{a_2} is i and that of L_j is m, with $I_i \subseteq I_m$ in the input, by Lemma 6.6.1, L_{a_2} dominates L_j. □

By Lemma 6.6.14, we let $\mathcal{L}$ be the union of the lists of S'_1 and the lists of S'_2 after and including b' (if $b' = c_2 + 1$, then $\mathcal{L} = S'_1$).

Observation 6.6.15. *All algorithm invariants hold on* $\mathcal{L}$.

Proof. As the analysis in Section 6.6.2, $S'_1 \cup S'_2$ must contain a canonical list of $\mathcal{I}[1, i]$. In light of Lemma 6.6.14(2), $\mathcal{L}$ also contains a canonical list.

Also, the values of $x(\mathcal{C}_L)$ for all lists L of S'_1 (resp., S'_2) are strictly increasing. By Lemma 6.6.14(1), the values of $x(\mathcal{C}_L)$ for all lists L of $\mathcal{L}$ are also strictly increasing. On the other hand, the values of $\delta(\mathcal{C}_L)$ for all lists L of S'_1 (resp., S'_2) are strictly decreasing. The definition of b' makes sure that the values of $\delta(\mathcal{C}_L)$ for all lists L of $\mathcal{L}$ must be strictly decreasing. Also, note that the lists of $\mathcal{L}$ whose last indices are i are all before the lists whose last indices are m.

Hence, all algorithm invariants hold on $\mathcal{L}$. □

The following lemma will be useful for the algorithm implementation.

Lemma 6.6.16. *For each list* $L_j \in \mathcal{L}$, *if* $L_j \neq L^*_c$, *then* $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$; *if* $L_j \notin \{L_{a_2}, L^*_c, L_{c_1}, L_{c_2}\}$, *then* $\delta(\mathcal{C}_{L_j}) = \delta(\mathcal{C}_{L'_j})$.

Proof. Consider any list $L_j \in \mathcal{L}$.

If $L_j \neq L^*_c$, then since $a_1 = 0$, $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$ always holds regardless whether the last index of L_j is i or m.

Assume $L_j \notin \{L_{a_2}, L^*_c, L_{c_1}, L_{c_2}\}$. To prove that $\delta(\mathcal{C}_{L_j}) = \delta(\mathcal{C}_{L'_j})$, if $j \leq b$, then we can apply the analysis in the proof of Lemma 6.6.6; otherwise, we can apply the analysis in the proof of Lemma 6.6.13. We omit the details. □

The third subcase $x^r_i < x^r_{m'}$.. In this case, for each list $L'_j \in \mathcal{L}'$, as analyzed in the proof of Lemma 6.6.3, only Case II of our preliminary algorithm happens, and thus L_j is obtained from L'_j by inserting i into L'_j right before the last index. Further, it holds that $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$ regardless of whether the last index of L'_j is m or m'. Since $x(\mathcal{C}_{L'_j})$ is strictly increasing on $j \in [1, a]$, $x(\mathcal{C}_{L_j})$ is also strictly increasing on $j \in [1, a]$.

Consider any list $L'_j \in \mathcal{L}'$ with $j \leq b$. Recall that the last index of L'_j is m'. By Observation 6.4.3, $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m', \mathcal{C}_{L_j})\}$, and $d(m', \mathcal{C}_{L_j}) = x^r_{m'}(\mathcal{C}_{L_j}) - x^r_{m'} = x(\mathcal{C}_{L_j}) - x^r_{m'}$. Thus, $d(m', \mathcal{C}_{L_j})$ strictly increases on $j \in [1, b]$. Since $\delta(\mathcal{C}_{L'_j})$ strictly decreases on $j \in [1, b]$, $\delta(\mathcal{C}_{L_j})$ is a unimodal function on $j \in [1, b]$ (i.e., it first strictly decreases and then strictly increases). If $\delta(\mathcal{C}_{L_1}) \leq \delta(\mathcal{C}_{L_2})$, then let $e_1 = 1$; otherwise, define e_1 to be the largest index $j \in [2, b]$ such that $\delta(\mathcal{C}_{L_{j-1}}) > \delta(\mathcal{C}_{L_j})$. Hence, $\delta(\mathcal{C}_{L_j})$ is strictly decreasing on $j \in [1, e_1]$.

Lemma 6.6.17. *If $e_1 < b$, then L_{e_1} dominates L_j for any $j \in [e_1 + 1, b]$.*

Proof. Assume $e_1 < b$ and let j be any index in $[e_1 + 1, b]$. By our definition of e_1 and since $\delta(\mathcal{C}_{L_j})$ is unimodal on $[1, b]$, it holds that $\delta(\mathcal{C}_{L_{e_1}}) \leq \delta(\mathcal{C}_{L_j})$. Recall that $x(\mathcal{C}_{L_{e_1}}) < x(\mathcal{C}_{L_j})$. Since the last indices of both L_{e_1} and L_j are m', by Lemma 6.6.2, L_{e_1} dominates L_j. □

Due to Lemma 6.6.17, let $S_1 = \{L_1, L_2, \ldots, L_{e_1}\}$.

Consider any list $L'_j \in \mathcal{L}'$ with $j > b$. Recall that the last index of L'_j is m. Similarly as above, $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L_j})\}$ and $d(m, \mathcal{C}_{L_j}) = x(\mathcal{C}_{L_j}) - x^r_m$. Similarly, $\delta(\mathcal{C}_{L_j})$ is a unimodal function on $j \in [b + 1, a]$. If $\delta(\mathcal{C}_{L_{b+1}}) \leq \delta(\mathcal{C}_{L_{b+2}})$, then we let $e_2 = b + 1$; otherwise, define e_2 to be the largest index $j \in [b + 1, a]$ such that $\delta(\mathcal{C}_{L_{j-1}}) > \delta(\mathcal{C}_{L_j})$. Hence, $\delta(\mathcal{C}_{L_j})$ is strictly decreasing on $j \in [b + 1, e_2]$. By a similar proof as Lemma 6.6.17, we can show that if $e_2 < a$, then L_{e_2} dominates L_j for any $j \in [e_2 + 1, a]$. Let $S_2 = \{L_{b+1}, L_{b+2}, \ldots, L_{e_2}\}$.

We finally combine S_1 and S_2 to obtain $\mathcal{L}$ as follows. Define b' to be the smallest index j of $[b + 1, e_2]$ such that $\delta(\mathcal{C}_{L_{e_1}}) > \delta(\mathcal{C}_{L_j})$, and if no such index exists, then let $b' = e_2 + 1$.

Lemma 6.6.18. *If $b' > b + 1$, then L_{e_1} dominates L_j of S_2 for any $j \in [b + 1, b' - 1]$.*

Proof. Assume $b' > b + 1$ and let j be any index in $[b + 1, b' - 1]$. Since $\delta(\mathcal{C}_{L_j})$ is strictly decreasing on $j \in [b + 1, e_2]$, by the definition of b', $\delta(\mathcal{C}_{L_{e_1}}) \leq \delta(\mathcal{C}_{L_j})$.

Recall that $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(m, \mathcal{C}_{L_j})\}$, $\delta(\mathcal{C}_{L_{e_1}}) = \max\{\delta(\mathcal{C}_{L'_{e_1}}), d(m', \mathcal{C}_{L_{e_1}})\}$, and $\delta(\mathcal{C}_{L'_j}) < \delta(\mathcal{C}_{L'_{e_1}})$. Hence, we obtain $\delta(\mathcal{C}_{L'_j}) < \delta(\mathcal{C}_{L'_{e_1}}) \leq \delta(\mathcal{C}_{L_j})$, and thus $\delta(\mathcal{C}_{L_j}) = d(m, \mathcal{C}_{L_j})$. Since $\delta(\mathcal{C}_{L_{e_1}}) \leq \delta(\mathcal{C}_{L_j})$, $\delta(\mathcal{C}_{L_{e_1}}) \leq d(m, \mathcal{C}_{L_j})$. Further, recall that $x(\mathcal{C}_{L_{e_1}}) < x(\mathcal{C}_{L_j})$. Then, Lemma 6.6.1 applies since the last index of L_{e_1} is m' and that of L_j is m, with $x^r_{m'} \leq x^r_m$. By Lemma 6.6.1, L_{e_1} dominates L_j. □

In light of Lemma 6.6.18, we let $\mathcal{L} = S_1 \cup \{L_{b'}, \ldots, L_{e_2}\}$ if $b' \neq e_2 + 1$ and $\mathcal{L} = S_1$ otherwise. By similar analysis as before, we can show that all algorithm invariants hold on $\mathcal{L}$, and we omit the details. The following lemma will be useful for the algorithm implementation.

Lemma 6.6.19. *For each list $L_j \in \mathcal{L}$, $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$; if $L_j \notin \{L_{e_1}, L_{e_2}\}$, then $\delta(\mathcal{C}_{L_j}) = \delta(\mathcal{C}_{L'_j})$.*

Proof. We have shown that $x(\mathcal{C}_{L_j}) = x(\mathcal{C}_{L'_j}) + |I_i|$ for any $j \in [1, a]$.

Consider any list $L_j \in \mathcal{L}$ and $j \notin \{e_1, e_2\}$. By the similar analysis as in Lemma 6.6.13, we can show that $\delta(\mathcal{C}_{L_j}) = \delta(\mathcal{C}_{L'_j})$. The details are omitted. □

6.6.3 The Algorithm Implementation

In this section, we implement our pruning algorithm described in Section 6.6.2 in $O(n \log n)$ time and $O(n)$ space. We first show how to compute the optimal value δ_{opt} and then show how to construct an optimal list L_{opt} in Section 6.6.4.

Since $\mathcal{L}$ may have $\Theta(n)$ lists and each list may have $\Theta(n)$ intervals, to avoid $\Omega(n^2)$ time, the key idea is to maintain the lists of $\mathcal{L}$ implicitly. We show that it is sufficient to maintain the "x-values" $x(\mathcal{C}_L)$ and the "δ-values" $\delta(\mathcal{C}_L)$ for all lists L of $\mathcal{L}$, as well as the list index b and the interval indices m' and m. To this end, and in particular, to update the x-values and the δ-values after each interval I_i is processed, our implementation heavily relies on Lemmas 6.6.6, 6.6.13, 6.6.16, and 6.6.19. Intuitively, these lemmas guarantee that although the x-values of all lists of $\mathcal{L}$ need to change, all but a constant number of them increase by the same amount, which can be updated implicitly in constant time; similarly, only a constant number of δ-values need to be updated. The details are given below.

Let $\mathcal{L} = \{L_1, L_2, \ldots, L_a\}$ such that $x(\mathcal{C}_{L_j})$ strictly increases on $j \in [1, a]$, and thus, $\delta(\mathcal{C}_{L_j})$ strictly decreases on $j \in [1, a]$ by the algorithm invariants.

We maintain a balanced binary search tree T whose leaves from left to right correspond to the ordered lists of $\mathcal{L}$. Let $v_1, \ldots, v_a$ be the leaves of T from left to right, and thus, v_j corresponds to L_j for each $j \in [1, a]$. For each $j \in [1, a]$, v_j stores a *δ-value* $\delta(v_j)$

that is equal to $\delta(\mathcal{C}_{L_j})$, and v_j stores another *x-value* $x(v_j)$ that is equal to $x(\mathcal{C}_{L_j}) - R$, where R is a *global shift* value maintained by the algorithm.

In addition, we maintain a pointer p_b pointing to the leaf $v(b)$ of T if $b \neq 0$ and $p_b = null$ if $b = 0$. We also maintain the interval indices m and m'. Again, if $p_b = null$, then m' is undefined.

Initially, after I_1 is processed, $\mathcal{L}$ consists of the single list $L = \{1\}$. We set $R = 0$, $m = 1$, and $p_b = null$. The tree T consists of only one leaf v_1 with $\delta(v_1) = 0$ and $x(v_1) = x_1^r$.

In general, we assume I_{i-1} has been processed and T, m, m', p_b, and R have been correctly maintained. In the following, we show how to update them for processing I_i. In particular, we show that processing I_i takes $O((k+1)\log n)$ time, where k is the number of lists removed from $\mathcal{L}$ during processing I_i. Since our algorithm will generate at most n new lists for $\mathcal{L}$ and each list will be removed from $\mathcal{L}$ at most once, the total time of the algorithm is $O(n \log n)$.

As in Section 6.6.2, we let $\mathcal{L}' = \{L'_1, L'_2, \ldots, L'_a\}$ denote the original set $\mathcal{L}$ before I_i is processed. Again, if $b \neq 0$, then $\mathcal{L}'_1 = \{L'_1, \ldots, L'_b\}$ and $\mathcal{L}'_2 = \{L'_{b+1}, \ldots, L'_a\}$. We consider the five subcases discussed in Section 6.6.2.

The Case $\mathcal{L}'_1 = \emptyset$

In this case, the last indices of all lists of $\mathcal{L}'$ are m.

The first subcase $x_m^r \leq x_i^r$.. In this case, in general we have $\mathcal{L} = \{L_j \mid a_1 \leq j \leq a_2\}$. We first find a_1 and remove the lists $L_1, \ldots, L_{a_1-1}$ if $a_1 > 1$ as follows.

Starting from the leftmost leaf v_1 of T, if $x(v_1) + R$ (which is equal to $x(\mathcal{C}_{L'_1})$) is larger than x_i^l, then $a_1 = 0$ and we are done. Otherwise, we consider the next leaf v_2. In general, suppose we are considering leaf v_j. If $x(v_j) + R > x_i^l$, then we stop with $a_1 = j - 1$. Otherwise, we remove leaf v_{j-1} (not v_j) from T and continue to consider the next leaf v_{j+1} if $j \neq a$ (if $j = a$, then we stop with $a_1 = a$).

If $a_1 \neq 0$, then the above has found the leaf v_{a_1}. In addition, we update $x(v_{a_1}) = x_i^r - R - |I_i|$ (we have minus $|I_i|$ here because later we will increase R by $|I_i|$).

Next we find a_2 and remove the lists $L_{a_2+1}, \ldots, L_a$ (by removing the corresponding leaves from T) if $a_2 < a$, as follows. Recall that for each $j \in [a_1 + 1, a]$, $\delta(\mathcal{C}_{L_j}) =$

$\max\{\delta(\mathcal{C}_{L'_j}), d(i, \mathcal{C}_{L_j})\}$, with $\delta(\mathcal{C}_{L'_j}) = \delta(v_j)$ and $d(i, \mathcal{C}_{L_j}) = x_i^l(\mathcal{C}_{L_j}) - x_i^l = x(\mathcal{C}_{L'_j}) - x_i^l = x(v_j) + R - x_i^l$. Hence, we have $\delta(\mathcal{C}_{L_j}) = \max\{\delta(v_j), x(v_j) + R - x_i^l\}$.

If $a_1 = a$, then we have $a_2 = a_1$ and we are done. Otherwise we do the following. Starting from the rightmost leaf v_a of T, we check whether $\max\{\delta(v_{a-1}), x(v_{a-1}) + R - x_i^l\} \leq \max\{\delta(v_a), x(v_a) + R - x_i^l\}$. If yes, we remove v_a from T and continue to consider v_{a-1}. In general, suppose we are considering v_j. If $j = a_1$, then we stop with $a_2 = a_1$. Otherwise, we check whether $\max\{\delta(v_{j-1}), x(v_{j-1})+R-x_i^l\} \leq \max\{\delta(v_j), x(v_j)+R-x_i^l\}$. If yes, we remove v_j from T and proceed on v_{j-1}. Otherwise, we stop with $a_2 = j$.

Suppose the above procedure finds leaf v_j with $a_2 = j$. We further update $\delta(v_j) = \max\{\delta(v_j), x(v_j) + R - x_i^l\}$. By Lemma 6.6.6, we do not need to update other δ-values.

The above has updated the tree T. In addition, we update $R = R + |I_i|$, which actually implicitly updates all x-values by Lemma 6.6.6. Finally, we update $m = i$ since the last indices of all updated lists of $\mathcal{L}$ are now i.

This finishes our algorithm for processing I_i. Clearly, the total time is $O((k + 1)\log n)$ since removing each leaf of T takes $O(\log n)$ time, where k is the number of leaves that have been removed from T.

The second subcase $x_m^r > x_i^r$.. In this case, roughly speaking, we should compute the set $\mathcal{L} = \{L_1, \ldots, L_{c_1}, L_{c'}, L_{c'+1}, \ldots, L_{c_2}\}$.

We first compute the index c, i.e., find the leaf v_c of T. This can be done by searching T in $O(\log n)$ time as follows. Note that for a list L'_j, to check whether $x_i^l > x_m^l(\mathcal{C}_{L'_j})$, since $x_m^l(\mathcal{C}_{L'_j}) = x(\mathcal{C}_{L'_j}) - |I_m| = x(v_j) + R - |I_m|$, it is equivalent to checking whether $x_i^l > x(v_j) + R - |I_m|$, which is equivalent to $x_i^l - R + |I_m| > x(v_j)$. Consequently, v_c is the rightmost leaf v of T such that $x_i^l - R + |I_m| > x(v)$, and thus v_c can be found by searching T in $O(\log n)$ time.

Next, we find c_1, and remove the leaves v_j with $j \in [c_1 + 1, c]$ if $c_1 < c$, as follows (note that if the above step finds $c = 0$, then we skip this step).

Recall that for each $j \in [1, c]$, $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L'_j}), d(i, \mathcal{C}_{L_j})\}$, with $\delta(\mathcal{C}_{L'_j}) = \delta(v_j)$ and $d(i, \mathcal{C}_{L_j}) = x_i^l(\mathcal{C}_{L_j}) - x_i^l = x(\mathcal{C}_{L'_j}) - x_i^l = x(v_j) + R - x_i^l$. Hence, we have $\delta(\mathcal{C}_{L_j}) = \max\{\delta(v_j), x(v_j) + R - x_i^l\}$.

Starting from v_c, we first check whether $\delta(\mathcal{C}_{L_{c-1}}) > \delta(\mathcal{C}_{L_c})$, by computing $\delta(\mathcal{C}_{L_{c-1}})$ and $\delta(\mathcal{C}_{L_c})$ as above. If yes, then $c_1 = c$ and we stop. Otherwise, we remove v_c and proceed on considering v_{c-1}. In general, suppose we are considering v_j. If $j = 1$, then we stop with $c_1 = 1$. Otherwise, we check whether $\delta(\mathcal{C}_{L_{j-1}}) > \delta(\mathcal{C}_{L_j})$. If yes, then $c_1 = j$; otherwise, we remove v_j and proceed on v_{j-1}.

In addition, after v_{c_1} is found as above, we update $\delta(v_{c_1}) = \max\{\delta(v_{c_1}), x(v_{c_1}) + R - x_i^l\}$.

Next, consider the new list L_c^*, which is $L_{c+0.5}$. We have the following $\delta(\mathcal{C}_{L_c^*}) = \max\{\delta(\mathcal{C}_{L_c'}), d(m, \mathcal{C}_{L_c^*})\} = \max\{\delta(\mathcal{C}_{L_c'}), x_m^l(\mathcal{C}_{L_c^*}) - x_m^l\}$. Due to that $\delta(\mathcal{C}_{L_c'}) = \delta(v_c)$ and $x_m^l(\mathcal{C}_{L_c^*}) = x_i^r$, we have $\delta(\mathcal{C}_{L_c^*}) = \max\{\delta(v_c), x_i^r - x_m^l\}$ (if the above has removed v_c, then we temporarily keep the value $\delta(v_c)$ before v_c is removed). Also, recall that $x(\mathcal{C}_{L_c^*}) = x_i^r + |I_m|$. Therefore, we can compute both $\delta(\mathcal{C}_{L_c^*})$ and $x(\mathcal{C}_{L_c^*})$ in constant time. We insert a new leaf $v_{c+0.5}$ to T corresponding to L_c^*, with $\delta(v_{c+0.5}) = \delta(\mathcal{C}_{L_c^*})$ and $x(v_{c+0.5}) = x(\mathcal{C}_{L_c^*}) - R - |I_i|$ (the minus $|I_i|$ is due to that later we will increase R by $|I_i|$).

Next, we determine c_2, and remove the leaves v_j with $j \in [c_2 + 1, a]$ if $c_2 < a$, as follows. Recall that for each $j \in [c+1, a]$, $\delta(\mathcal{C}_{L_j}) = \max\{\delta(\mathcal{C}_{L_j'}), d(m, \mathcal{C}_{L_j})\}$, with $\delta(\mathcal{C}_{L_j'}) = \delta(v_j)$ and $d(m, \mathcal{C}_{L_j}) = x_m^r(\mathcal{C}_{L_j}) - x_m^r = x(\mathcal{C}_{L_j'}) + |I_i| - x_m^r = x(v_j) + R + |I_i| - x_m^r$. Hence, we have $\delta(\mathcal{C}_{L_j}) = \max\{\delta(v_j), x(v_j) + R + |I_i| - x_m^r\}$, which can be computed in constant time once we access the leaf v_j.

Starting from the rightmost leaf v_a, in general, suppose we are considering a leaf v_j. If $j = c + 0.5$, then we stop with $c_2 = c + 0.5$. Otherwise, let v_h be the left neighboring leaf of v_j (so h is either $j - 1$ or $j - 0.5$). We check whether $\delta(\mathcal{C}_{L_h}) > \delta(\mathcal{C}_{L_j})$ (the two values can be computed as above). If yes, we stop with $c_2 = j$; otherwise, we remove v_j from T and proceed on considering v_h.

If the above procedure returns $c_2 \geq c + 1$, then we further check whether $x(\mathcal{C}_{L_c^*}) = x(\mathcal{C}_{L_{c+1}})$. If yes, then we remove the leaf $v_{c+0.5}$ from T. If $c_2 \geq c + 1$, we also need to update $\delta(v_{c_2}) = \max\{\delta(v_{c_2}), x(v_{c_2}) + R + |I_i| - x_m^r\}$.

Finally, we determine c' and remove all leaves strictly between v_{c_1} and $v_{c'}$, as follows. Recall that given any leaf v_j of T, we can compute $\delta(\mathcal{C}_{L_j})$ in constant time. Starting from the right neighboring leaf of v_{c_1}, in general, suppose we are considering a leaf v_j. If $\delta(\mathcal{C}_{L_{c_1}}) \leq \delta(\mathcal{C}_{L_j})$, then we remove v_j and proceed on the right neighboring leaf of v_j. This procedure continues until either $\delta(\mathcal{C}_{L_{c_1}}) > \delta(\mathcal{C}_{L_j})$ or v_j is the rightmost leaf and has been removed.

In addition, we update $R = R + |I_i|$. In light of Lemma 6.6.13 and by our way of setting the value $x(v_{c+0.5})$, this updates all x-values. Also, the above has "manually" set the values $\delta(v_{c_1})$, $\delta(v_{c_2})$, and $\delta(v_{c_{c+0.5}})$, by Lemma 6.6.13, all δ-values have been updated. Finally, we update m, m', and p_b as follows.

In the general case where $1 \leq c < a$ and $c' \neq c_2 + 1$, we set $m' = i$ and p_b to the leaf v_{c_1}. If $c' = c_2 + 1$, then the last indices of all lists of $\mathcal{L}$ are i, and thus we set $m = i$ and $p_b = null$. If $c = 0$, then the last indices of all lists of $\mathcal{L}$ are m, then we do not need to update anything. If $c = a$, then if $L_c^* \notin \mathcal{L}$, then the last indices of all lists of $\mathcal{L}$ are i and we set $m = i$ and $p_b = null$, and if $L_c^* \in \mathcal{L}$, then we set $m' = i$ and p_b to v_{c_1}.

This finishes processing I_i. The total time is again as claimed before.

The Case $\mathcal{L}_1' \neq \emptyset$

In this case, $\mathcal{L}_1' = \{L_1', \ldots, L_b'\}$ and $\mathcal{L}_2' = \{L_{b+1}', \ldots, L_a'\}$. The last indices of all lists of $\mathcal{L}_1'$ (resp., $\mathcal{L}_2'$) are m' (resp., m). Note that the pointer p_b points to the leaf v_b.

The first subcase $x_i^r \geq x_{m}^r$.. In this case, the implementation is similar to the first subcase of Section 6.6.3, so we omit the details.

The second subcase $x_{m'}^r \leq x_i^r < x_m^r$.. As our algorithm description in Section 6.6.2, we first apply the similar implementation as the first subcase of Section 6.6.3 on the leaves from v_1 to v_b, and then apply the similar implementation as the second subcase of Section 6.6.3 on the leaves from v_{b+1} to v_a. So the leaves of the current tree corresponding to the lists in $S_1' \cup S_2'$, i.e., $\{L_1 \ldots, L_{a_2}, L_{b+1}, \ldots, L_{c_1}, L_{c'}, \ldots, L_{c_2}\}$, as defined in the second subcase of Section 6.6.2.

Next, we determine b' and remove all leaves from T strictly between v_{a_2} and $v_{b'}$. Starting from the right neighboring leaf of v_{a_2}, in general, suppose we are considering a leaf v_j. If $\delta(\mathcal{C}_{L_{a_2}}) \leq \delta(\mathcal{C}_{L_j})$ (as before, these two values can be computed in constant time once we have access to v_{a_2} and v_j), then we remove v_j and proceed on the right neighboring leaf of v_j. This procedure continues until either $\delta(\mathcal{C}_{L_{a_2}}) > \delta(\mathcal{C}_{L_j})$ or v_j is the rightmost leaf and has been removed.

Finally, we update $R = R + |I_i|$. To update p_b, m, and m', depending on the values c, c' and b', there are various cases. In the general case where $b + 1 \leq c < a$, $c' \neq c_2 + 1$, and $b' \neq c_2 + 1$, we update $p_b = v_{c_1}$ and $m' = i$. We omit the discussions for other special cases.

The third subcase $x_i^r < x_{m'}^r$.. In this case, starting from v_b, we first remove all leaves from v_{e_1+1} to v_b. The algorithm is very similar as before and we omit the details. Then, starting from v_a, we remove all leaves from v_{e_2+1} to v_a. Finally, starting from v_{e_1}, we remove all leaves strictly between v_{e_1} to $v_{b'}$. In addition, we update $R = R + |I_i|$. In the general case where $b' \neq e_2 + 1$, we set p_b pointing to leaf v_{e_1}; otherwise, we set $m = m'$ and $p_b = null$.

This finishes processing I_i for all five subcases. The algorithm finishes once I_n is processed, after which $\delta_{opt} = \delta(v)$, where v is the rightmost leaf of T (as $\delta(v)$ is the smallest among all leaves of T). Again, the total time of the algorithm is $O(n \log n)$. Clearly, the space used by our algorithm is $O(n)$.

6.6.4 Computing an Optimal List

As discussed above, after I_n is processed, the list (denoted by L_{opt}) corresponding to the rightmost leaf (denoted by v_{opt}) of T is an optimal list, and $\delta_{opt} = \delta(v_{opt})$. However, since our algorithm does not maintain the list L_{opt} explicitly, L_{opt} is not available after the algorithm finishes. In this section, we give a way (without changing the complexity asymptotically) to maintain more information during the algorithm such that after it finishes, we can reconstruct L_{opt} in additional $O(n)$ time.

We first discuss some intuition. Consider a list $L \in \mathcal{L}$ before interval I_i is processed. During processing I_i for L, observe that the position of i in the updated list L is uniquely determined by the input position of the last interval I_m of L (i.e., depending on whether $x_i^r \geq x_m^r$). However, uncertainty happens when L generates another "new" list L^*. More specifically, suppose L is a canonical list of $\mathcal{I}[1, i-1]$. If there is no new list L^*, then by our observations (i.e., Lemmas 6.4.2 and 6.4.4), the updated L is a canonical list of $\mathcal{I}[1, i]$. Otherwise, we know (by Lemma 6.4.6) that one of L and L^* is a canonical list of $\mathcal{I}[1, i]$, but we do not know exactly which one is. This is where the uncertainty happens and indeed this is why we need to keep both L and L^* (thanks to Lemma 6.4.7, we only need to keep one such new list). Therefore, in order to reconstruct L_{opt}, if processing I_i generates a new list L^* in $\mathcal{L}$, then we need to keep the relevant information about L^*.

The details are given below.

Specifically, we maintain an additional binary tree T' (not a search tree). As in T, the leaves of T' from left to right correspond to the ordered lists of $\mathcal{L}$. Consider a leaf v of T' that corresponds to a list $L \in \mathcal{L}$. Suppose after processing I_i, L generates a new list L^* in $\mathcal{L}$. Let m be the last index of the original L (before I_i is processed). According to our algorithm, we know that the last two indices of the updated L are m and i with i as the last index and the last two indices of L^* are i and m with m as the last index. Correspondingly, we update the tree T' as follows. First, we store i at v, e.g., by setting $A(v) = i$, which means that there are two choices for processing I_i. Second, we create two children v_1 and v_2 for v and they correspond to the lists L and L^*, respectively. Thus, v now becomes an internal node. Third, on the new edge (v, v_1), we store an ordered pair (m, i), meaning that m is before i in L; similarly, on the edge (v, v_2), we store the pair (i, m). In this way, each internal node of T' stores an interval index and each edge of T' stores an ordered pair.

After the algorithm finishes, we reconstruct the list L_{opt} in the following way. Let π be the path from the root to the rightmost leaf v_{opt} of T'. We will construct L_{opt} by considering all intervals from I_1 to I_n and simultaneously considering the nodes in π. Initially, let $L_{opt} = \{1\}$. Then, we consider I_2 and the first node of π (i.e., the root of T'). In general, suppose we are considering I_i and a node v of π. We first assume that v is an internal node (i.e., $v \neq v_{opt}$).

If $i < A(v)$, then only Case I or Case II of our preliminary algorithm happens, and we insert i into L_{opt} based on whether $x_i^r \geq x_m^r$ (specifically, if $x_i^r \geq x_m^r$, then we append i at the end of L_{opt}; otherwise, we insert i right before the last index of L_{opt}) and then proceed on I_{i+1}.

If $i \geq A(v)$ (in fact, i must be equal to $A(v)$), then we insert i into L_{opt} based on the ordered pair of the next edge of v in π (specifically, if i is at the second position of the pair, then i is appended at the end of L_{opt}; otherwise, i is inserted right before the last index of L_{opt}) and then proceed on the next node of π and I_{i+1}.

If $v = v_{opt}$, then we insert i into L_{opt} based on whether $x_i^r \geq x_m^r$ as above, and then proceed on I_{i+1}. The algorithm finishes once I_n is processed, after which L_{opt} is constructed. It is easy to see that the algorithm runs in $O(n)$ time and $O(n)$ space.

Once L_{opt} is computed, we can apply the left-possible placement strategy to compute an optimal configuration in additional $O(n)$ time.

Theorem 6.6.20. *Given a set of n intervals on a line, the interval separation problem is solvable in $O(n \log n)$ time and $O(n)$ space.*

6.7 The Lower Bound

By a linear-time reduction from the integer element distinctness problem [72,73], we can obtain an $\Omega(n \log n)$ time lower bound for the problem under the algebraic decision tree model, which implies the optimality of our algorithm.

Given a set of n integers $A = \{a_1, a_2, \ldots, a_n\}$, the element distinctness problem is to ask whether there are two elements of A that are equal. The problem has an $\Omega(n \log n)$ time lower bound under the algebraic decision tree model [72, 73]. We create a set $\mathcal{I}$ of n intervals as an instance of our interval separation problem as follows. For each $a_i \in A$, we create an interval I_i centered at a_i with length 0.1. Let $\mathcal{I}$ be the set of all intervals. Since all elements of A are integers, it is easy to see that no two elements of A are equal if and only if no two intervals of $\mathcal{I}$ intersect. On the other hand, no two intervals of $\mathcal{I}$ intersect if and only if the optimal value δ_{opt} in our interval separation problem on $\mathcal{I}$ is equal to zero. This completes the reduction. This reduction actually shows that even if all intervals have the same length, the interval separation problem still has an $\Omega(n \log n)$ time lower bound.

CHAPTER 7

Future Work

We discuss some future work that are natural extensions of the problems we studied before.

1. The Cycle Version of Multiple Barrier Coverage Problem

In Chapter 5, we solved the line version of multiple barrier coverage problem. One natural extension is the cycle version of this problem. In the cycle version, there is a cycle C and each barrier becomes an arc on C. Each sensor is still a point on C but now can cover an arc of C centered at the sensor. In addition, in the cycle version, the distance of any two points on C is defined as the length of the shortest path between the two points on C.

If there is only one barrier, i.e., $m = 1$, the problem has already been studied and a linear time algorithm is known [33] after the sensors are sorted on C. For the general value m, however, the problem has not been considered before. We propose to study this problem and try to extend our algorithm for the line version in Chapter 5.

2. The Min-Sum Version of the Multiple Barrier Coverage Problem

In Chapter 5, we studied the min-max version of multiple barrier coverage problem. A closely related problem is the min-sum version, defined as follows. Given m barriers and n sensors on a line L, the goal is to move sensors so that the union of the covering intervals of all sensors covers all barriers and the total sum of the movements of all sensors is minimized.

If $m = 1$, i.e., there is only one barrier, the problem has been studied before and the previously best algorithm runs in $O(n \log n)$ time [63]. To our best knowledge, the general case has not been considered before. We plan to study this problem. One possible direction is to see whether the algorithm in [63] can somehow be extended.

It may also be interesting to consider the cycle version of the problem.

3. Separating Overlapped Intervals on a Cycle

We solve the separating overlapped intervals problem on a line in Chapter 6. It might also be interesting to consider the cycle version of the problem. Let $\mathcal{I}$ be a set of n intervals on a closed cycle C. We say that two intervals *overlap* if their intersection contains more than one point. The problem is to move the intervals of $\mathcal{I}$ along the cycle C such that no two intervals overlap and the maximum moving distance of these intervals is minimized.

If all intervals of $\mathcal{I}$ have the same length, then the problem is essentially the same as the problem of spreading points on a cycle that we studied in Chapter 3, and thus, by using our algorithm in Chapter 3, after the left endpoints of the intervals are sorted, the problem can be solved in $O(n)$ time. For the general problem where intervals may have different lengths, to the best of our knowledge, the problem has not been studied before.

REFERENCES

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

[2] S. Devadoss and J. O'Rourke, *Discrete and Computational Geometry.* New Jersey: Princeton University Press, 2011.

[3] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, 2nd ed. Springer-Verlag, 1988.

[4] J. Sack and J. Urrutia, Eds., *Handbook of Computational Geometry.* Elsevier, Amsterdam, The Netherlands, 2000.

[5] M. Chrobak, C. Dürr, W. Jawor, L. Kowalik, and M. Kurowski, "A note on scheduling equal-length jobs to maximize throughput," *Journal of Scheduling*, vol. 9(1), pp. 71–73, 2006.

[6] M. Garey, D. Johnson, B. Simons, and R. Tarjan, "Scheduling unit-time tasks with arbitrary release times and deadlines," *SIAM Journal of Computing*, vol. 10, pp. 256–269, 1981.

[7] J. Kleinberg and E. Tardos, *Algorithm Design.* Boston, MA, USA: Addison-Wesley, 2005.

[8] T. Lang and E. Fernández, "Scheduling of unit-length independent tasks with execution constraints," *Information Processing Letters*, vol. 4, pp. 95–98, 1976.

[9] E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys, Sequencing and scheduling: Algorithms and complexity, in *Handbooks in Operations Research and Management Science 4,* S.C. Graves, A.H.G. Rinnooy Kan and P.H. Zipkin (eds.). Elsevier, 1993.

[10] B. Simons, "A fast algorithm for single processor scheduling," in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, 1978, pp. 246–252.

[11] N. Vakhania and F. Werner, "Minimizing maximum lateness of jobs with naturally bounded job data on a single machine in polynomial time," *Theoretical Computer Science*, vol. 501, pp. 72–81, 2013.

[12] A. Bar-Noy, D. Rawitz, and P. Terlecky, "Maximizing barrier coverage lifetime with mobile sensors," in *Proc. of the 21st European Symposium on Algorithms (ESA)*, 2013, pp. 97–108.

[13] B. Bhattacharya, B. Burmester, Y. Hu, E. Kranakis, Q. Shi, and A. Wiese, "Optimal movement of mobile sensors for barrier coverage of a planar region," *Theoretical Computer Science*, vol. 410, no. 52, pp. 5515–5528, 2009.

[14] D. Chen, X. Tan, H. Wang, and G. Wu, "Optimal point movement for covering circular regions," *Algorithmica*, vol. 72, pp. 379–399, 2013.

[15] M. Li, X. Sun, and Y. Zhao, "Minimum-cost linear coverage by sensors with adjustable ranges," in *Proc. of the 6th International Conference on Wireless Algorithms, Systems, and Applications*, 2011, pp. 25–35.

[16] M. Mehrandish, "On routing, backbone formation and barrier coverage in wireless ad doc and sensor networks," Ph.D. dissertation, Concordia University, Montreal, Quebec, Canada, 2011.

[17] M. Mehrandish, L. Narayanan, and J. Opatrny, "Minimizing the number of sensors moved on line barriers," in *Proc. of IEEE Wireless Communications and Networking Conference (WCNC)*, 2011, pp. 653–658.

[18] W. Cao, J. Li, S. Li, and H. Wang, "Balanced splitting on weighted intervals," *Operations Research Letters*, vol. 43, pp. 396–400, 2015.

[19] S. Li and H. Wang, "Algorithms for minimizing the movements of spreading points in linear domains," in *Proc. of the 27th Canadian Conference on Computational Geometry (CCCG)*, 2015.

[20] ——, "Dispersing points on intervals," in *Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC)*, 2016, pp. 52:1–52:12.

[21] ——, "Dispersing points on intervals," *Discrete Applied Mathematics*, vol. 239, pp. 106–118, 2018.

[22] ——, "Algorithms for covering multiple barriers," in *Proceedings of the 15th Algorithms and Data Structures Symposium (WADS)*, 2017, pp. 533–544.

[23] W. Le, F. Li, Y. Tao, and R. Christensen, "Optimal splitters for temporal and multiversion databases," in *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 109–120.

[24] J. Chuzhoy, R. Ostrovsky, and Y. Rabani, "Approximation algorithms for the job interval selection problem and related scheduling problems," *Mathematics of Operations Research*, vol. 31, pp. 730–738, 2006.

[25] J. M. Keil, "On the complexity of scheduling tasks with discrete starting times," *Operations Research Letters*, vol. 12, pp. 293–295, 1992.

[26] K. Nakajima and S. Hakimi, "Complexity results for scheduling tasks with discrete starting times," *Journal of Algorithms*, vol. 3, pp. 344–361, 1982.

[27] F. Spieksma, "On the approximability of an interval scheduling problem," *Journal of Scheduling*, vol. 2, pp. 215–227, 1999.

[28] M. Golumbic, *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

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

[30] K. Ross and J. Cieslewicz, "Optimal splitters for database partitioning with size bounds," in *Proc. of the 12th International Conference on Database Theory (ICDT)*, 2009, pp. 98–110.

[31] S. Khanna, S. Muthukrishnan, and S. Skiena, "Efficient array partitioning," in *Proc. of the 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, 1997, pp. 616–626.

[32] S. Muthukrishnan and T. Suel, "Approximation algorithms for array partitioning problems," *Journal of Algorithms*, vol. 54, pp. 85–104, 2005.

[33] D. Chen, Y. Gu, J. Li, and H. Wang, "Algorithms on minimizing the maximum sensor movement for barrier coverage of a linear domain," *Discrete and Computational Geometry*, vol. 50, pp. 374–408, 2013.

[34] A. Dumitrescu and M. Jiang, "Constrained k-center and movement to independence," *Discrete Applied Mathematics*, vol. 159, pp. 859–865, 2011.

[35] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, pp. 373–395, 1984.

[36] L. G. Khachiyan, "Polynomial algorithm in linear programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, pp. 53–72, 1980.

[37] E. Demaine, M. Hajiaghayi, H. Mahini, A. Sayedi-Roshkhar, S. Oveisgharan, and M. Zadimoghaddam, "Minimizing movement," *ACM Transactions on Algorithms*, vol. 5, no. 30, 2009.

[38] J. Fiala, J. Kratochvíl, and A. Proskurowski, "Systems of distant representatives," *Discrete Applied Mathematics*, vol. 145, pp. 306–316, 2005.

[39] S. Cabello, "Approximation algorithms for spreading points," *Journal of Algorithms*, vol. 62, pp. 49–73, 2007.

[40] ref:DumitrescuDi12, "Dispersion in disks," *Theory of Computing Systems*, vol. 51, pp. 125–142, 2012.

[41] S. Ravi, D. Rosenkrantz, and G. Tayi, "Heuristic and special case algorithms for dispersion problems," *Operations Research*, vol. 42, no. 2, pp. 299–310, 1994.

[42] D. Wang and Y.-S. Kuo, "A study on two geometric location problems," *Information Processing Letters*, vol. 28, pp. 281–286, 1988.

[43] B. Chandra and M. Halldórsson, "Approximation algorithms for dispersion problems," *Journal of Algorithms*, vol. 38, pp. 438–465, 2001.

[44] Z. Friggstad and M. Salavatipour, "Minimizing movement in mobile facility location problems," *ACM Transactions on Algorithms*, vol. 7, 2011, article No. 28.

[45] E. Fernández, J. Kalcsics, and S. Nickel, "The maximum dispersion problem," *Omega*, vol. 41(4), pp. 721–730, 2013.

[46] G. Jäger, A. Srivastav, and K. Wolf, "Solving generalized maximum dispersion with linear programming," in *Proceedings of the 3rd International Conference on Algorithmic Aspects in Information and Management*, 2007, pp. 1–10.

[47] O. Prokopyev, N. Kong, and D. Martinez-Torres, "The equitable dispersion problem," *European Journal of Operational Research*, vol. 197(1), pp. 59–67, 2009.

[48] S. Ravi, D. Rosenkrantz, and G. Tayi, "Facility dispersion problems: Heuristics and special cases," *Algorithms and Data Structures*, vol. 519, pp. 355–366, 1991.

[49] C. Baur and S. Fekete, "Approximation of geometric dispersion problems," *Algorithmica*, vol. 30, no. 3, pp. 451–470, 2001.

[50] M. Benkert, J. Gudmundsson, C. Knauer, R. van Oostrum, and A. Wolff, "A polynomial-time approximation algorithm for a geometric dispersion problem," *Int. J. Comput. Geometry Appl.*, vol. 19, no. 3, pp. 267–288, 2009.

[51] E. Erkut, "The discrete p-dispersion problem," *European Journal of Operational Research*, vol. 46, pp. 48–60, 1990.

[52] R. Fowler, M. Paterson, and S. Tanimoto, "Optimal packing and covering in the plane are NP-complete," *Information Processing Letters*, vol. 12, pp. 133–137, 1981.

[53] Z. Füredi, "The densest packing of equal circles into a parallel strip," *Discrete and Computational Geometry*, vol. 6, pp. 95–106, 1991.

[54] C. Maranasa, C. Floudas, and P. Pardalosb, "New results in the packing of equal circles in a square," *Discrete Mathematics*, vol. 142, pp. 287–293, 1995.

[55] M. Chrobak, W. Jawor, J. Sgall, and T. Tichý, "Online scheduling of equal-length jobs: Randomization and restarts help," *SIAM Journal of Computing*, vol. 36(6), pp. 1709–1728, 2007.

[56] N. Vakhania, "A study of single-machine scheduling problem to maximize throughput," *Journal of Scheduling*, vol. 16, no. 4, pp. 395–403, 2013.

[57] S. Li and H. Shen, "Minimizing the maximum sensor movement for barrier coverage in the plane," in *Proc. of the 2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 244–252.

[58] S. Kumar, T. Lai, and A. Arora, "Barrier coverage with wireless sensors," in *Proc. of the 11th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2005, pp. 284–298.

[59] S. Dobrev, S. Durocher, M. Eftekhari, K. Georgiou, E. Kranakis, D. Krizanc, L. Narayanan, J. Opatrny, S. Shende, and J. Urrutia, "Complexity of barrier coverage with relocatable sensors in the plane," *Theoretical Computer Science*, vol. 579, pp. 64–73, 2015.

[60] J. Czyzowicz, E. Kranakis, D. Krizanc, I. Lambadaris, L. Narayanan, J. Opatrny, L. Stacho, J. Urrutia, and M. Yazdani, "On minimizing the maximum sensor movement for barrier coverage of a line segment," in *Proc. of the 8th International Conference on Ad-Hoc, Mobile and Wireless Networks*, 2009, pp. 194–212.

[61] H. Wang and X. Zhang, "Minimizing the maximum moving cost of interval coverage," in *Proc. of the 26th International Symposium on Algorithms and Computation (ISAAC)*, 2015, pp. 188–198, full version to appear in *International Journal of Computational Geometry and Application (IJCGA).*

[62] J. Czyzowicz, E. Kranakis, D. Krizanc, I. Lambadaris, L. Narayanan, J. Opatrny, L. Stacho, J. Urrutia, and M. Yazdani, "On minimizing the sum of sensor movements for barrier coverage of a line segment," in *Proc. of the 9th International Conference on Ad-Hoc, Mobile and Wireless Networks*, 2010, pp. 29–42.

[63] A. Andrews and H. Wang, "Minimizing the aggregate movements for interval coverage," *Algorithmica*, vol. 78, pp. 47–85, 2017.

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

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

[66] G. Frederickson, "Optimal algorithms for tree partitioning," in *Proc. of the 2nd Annual ACM-SIAM Symposium of Discrete Algorithms (SODA)*, 1991, pp. 168–177.

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

[68] M. Atallah, "Some dynamic computational geometry problems," *Computers and Mathematics with Applications*, vol. 11, pp. 1171–1181, 1985.

[69] J. Hershberger, "Finding the upper envelope of n line segments in $O(n \log n)$ time," *Information Processing Letters*, vol. 33, pp. 169–174, 1989.

[70] M. Sharir and P. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications.* New York: Cambridge University Press, 1996.

[71] J. Kleinberg and E. Tardos, *Algorithm Design.* Boston, MA, USA: Addison-Wesley, 2005, ch. 4.

[72] A. Lubiw and A. Rácz, "A lower bound for the integer element distinctness problem," *Information and Computation*, vol. 94, pp. 83–92, 1991.

[73] A. Yao, "Lower bounds for algebraic computation trees with integer inputs," *SIAM Journal on Computing*, vol. 20, pp. 655–668, 1991.

CURRICULUM VITAE

Shimin Li

EDUCATION

Ph.D., Computer Science. Utah State University, Logan, Utah. Expected in May 2018.

M.S., Control Science and Engineering. East China University of Science and Technology, Shanghai, China. July 2010.

B.S., Electrical Engineering and Information. Northeast Petroleum University, Daqing, China. June 2006.

RESEARCH INTERESTS

Algorithms and Theory, Data Structures, Computational Geometry, Combinatorial Optimization, Scheduling, Operations Research, etc.

BOOKS

Shimin Li *Java Programming is Easy*, Beijing: Tsinghua University Press, 2012.

JOURNAL PUBLICATIONS

Shimin Li and Haitao Wang. "Algorithms for Covering Multiple Barriers", submitted to *Theoretical Computer Science (TCS)*, 2017. (**Under Review**)

Shimin Li and Haitao Wang. "Algorithms for Minimizing the Movements of Spreading Points in Linear Domains", submitted to *Theoretical Computer Science (TCS)*, 2015. (**Under Review**)

Shimin Li and Haitao Wang. "Dispersing Points on Intervals", *Discrete Applied Mathematics*, Vol. 239, pages 106-118, 2018.

Wei Cao, Jian Li, Shimin Li, and Haitao Wang. "Balanced Splitting on Weighted Intervals", *Operations Research Letters*, Vol. 43, pages 396-400, 2015.

CONFERENCE PUBLICATIONS

Shimin Li and Haitao Wang. “Separating Overlapped Intervals on a Line”, submitted to *the 24th International Computing and Combinatorics Conference (COCOON'18)*, Qingdao, China, July 2018. (**Under Review**)

Shimin Li and Haitao Wang. “Algorithms for Covering Multiple Barriers”, *Proceedings of the 15th Algorithms and Data Structures Symposium (WADS)*, St. John's, Canada, August 2017, pages 533-544.

Shimin Li and Haitao Wang. “Dispersing Points on Intervals”, *Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC)*, Sydney, Australia, December 2016, pages 52:1-52:12.

Shimin Li and Haitao Wang. “Algorithms for Minimizing the Movements of Spreading Points in Linear Domains”, *Proceedings of the 26th Canadian Conference on Computational Geometry (CCCG)*, Kingston, Canada, August 2015, pages 187-192.

Shimin Li and Xingyu Wang. “A Modeling and Optimizing Method Based on the Topology Information of Wireless Sensor Network”, *Proceedings of the 29th Chinese Control Conference (CCC)*, Beijing, China, July 2010, pages 4802-4806.