Dispersing Points on Intervals*†

Shimin Li[1] and Haitao Wang[2]

1 **Department of Computer Science, Utah State University, Logan, UT 84322, USA**
shiminli@aggiemail.usu.edu

2 **Department of Computer Science, Utah State University, Logan, UT 84322, USA**
haitao.wang@usu.edu

Abstract

We consider a problem of dispersing points on disjoint intervals on a line. 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. Based on a greedy strategy, 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.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, I.1.2 Algorithms, I.3.5 Computational Geometry and Object Modeling

Keywords and phrases dispersing points, intervals, min-max, algorithms, cycles

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.52

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., [6, 10, 13, 14, 15, 19].

In this paper, we consider problems of dispersing points on intervals in linear domains including lines and cycles. 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 paper we present an $O(n)$ time algorithm for the 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.

1.1 Related Work

To the best of our knowledge, we have not found any previous work on the two problems studied in this paper. 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 [1] studied the problems of distributing a number of points within a polygonal

* A full version of the paper is available at https://arxiv.org/abs/1611.09485.
† This research was supported in part by NSF under Grant CCF-1317143.

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 [19] 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 [19] that both problems in 2D are NP-hard but can be solved efficiently in 1D. Refer to [2, 5, 7, 8, 12] for other geometric dispersion problems. Dispersion problems in various non-geometric settings were also considered [6, 10, 13, 14, 15]. These problems are in general NP-hard; approximation and heuristic algorithms were proposed for them.

On the other hand, problems on intervals usually have many applications. 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. [9] 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 [9] for this problem. The optimization version of the above problem was also studied [4, 17], where the goal is to find a maximum number of intervals that contain non-intersecting unit-length sub-intervals. Chrobak et al. [4] gave an $O(n^5)$ time algorithm for the problem, and later Vakhania [17] improved the algorithm to $O(n^2 \log n)$ time. The online version of the problem was also considered [3]. Other optimization problems on intervals have also been considered, e.g., see [9, 11, 16, 18].

1.2 Our Approaches

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" $\mathcal{L}$ 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, we convert it 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. 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 2. The cycle version is discussed in Section 3. Due to the space limit, some proofs are omitted but can be found in the full version of the paper.

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 2.1, we discuss some observations. In Section 2.2, we give an overview of our algorithm. The algorithm details are presented in Section 2.3. Finally, we discuss the correctness and analyze the running time in Section 2.4.

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

▶ **Corollary 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 $1 \leq i < j \leq n$. Then the solution is an optimal solution.*

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.

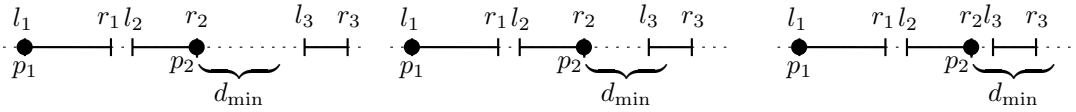

Figure 1 Illustrating the three cases when I_3 is being processed.

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" $\mathcal{L}$ of interval indices, which bears some important properties that eventually help us implement our algorithm in $O(n)$ time.

In fact, our algorithm is fairly simple. The most "complicated" part is to use a linked list to store $\mathcal{L}$ so that the following three operations on $\mathcal{L}$ 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 $\mathcal{L} = \{1\}$.

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 $\mathcal{L}$. 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. 1).

1. If $p_2 + d_{\min} \le l_3$, we set $p_3 = l_3$. We reset $\mathcal{L}$ to $\{3\}$. None of $d_{\min}$, i^*, and j^* needs to be changed in this case.
2. If $l_3 < p_2 + d_{\min} \le r_3$, we set $p_3 = p_2 + d_{\min}$. None of $d_{\min}$, i^*, and j^* needs to be changed. Further, the critical list $\mathcal{L}$ is updated as follows.
   We first give some "motivation" on why we need to update $\mathcal{L}$. 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

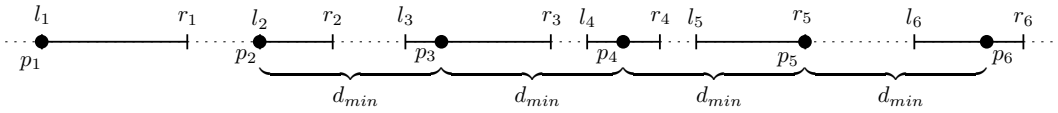

Figure 2 Illustrating the solution computed by our algorithm, with $i^* = 2$ and $j^* = 5$.

$d_{\min}$ 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 $\mathcal{L}$ as follows. If $l_2 - l_1 > \frac{l_3 - l_1}{2}$, we add 3 to the rear of $\mathcal{L}$. Otherwise, we first remove 2 from the rear of $\mathcal{L}$ 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 $\mathcal{L}$ and add 3 to the rear of $\mathcal{L}$, after which $\mathcal{L} = \{2, 3\}$.
   - **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 $\mathcal{L}$ 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 $\mathcal{L}$; otherwise, we remove 2 from $\mathcal{L}$ 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. 2 illustrates the solution found by our algorithm on six intervals.

2.3 The Algorithm

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

Initially we set $p_1 = l_1$, $d_{\min} = \infty$, $i^* = j^* = 1$, and $\mathcal{L} = \{1\}$. Suppose interval $i - 1$ has just been processed for some $i > 1$. Let the current critical list be $\mathcal{L} = \{k_s, k_{s+1}, \ldots k_t\}$ with $1 \le k_s < k_{s+1} < \cdots < k_t \le i - 1$, i.e., $\mathcal{L}$ 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 $1 \leq j \leq k_s$, 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 $k_s + 1 \leq j \leq i-1$, 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 $\mathcal{L}$ has the following *priority property*: If $\mathcal{L}$ has more than one element (i.e., $s < t$), then for any h with $s \leq h \leq t-1$, Inequality (1) holds for any j with $k_h + 1 \leq j \leq i-1$ and $j \neq k_{h+1}$.

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

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 (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 (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 1.** *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 (1), we have

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

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 (2) will obtain the observation. ◀

▶ Remark. By Corollary 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.

2.3.1 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 $\mathcal{L} = \{i\}$. The proof of Lemma 3 is omitted.

▶ **Lemma 3.** *In the case $p_{i-1} + d_{\min} \leq l_i$, all algorithm invariants hold after I_i is processed.*

2.3.2 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 $\mathcal{L}$ by the following *rear-processing procedure* (because the elements of $\mathcal{L}$ are considered from the rear to the front).

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

$$\frac{l_{k_t} - l_{k_{t-1}}}{k_t - k_{t-1}} > \frac{l_i - l_{k_{t-1}}}{i - k_{t-1}}. \tag{3}$$

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

If it is not true, then we remove k_t from $\mathcal{L}$ and decrease t by 1. Next, we continue to check whether Inequality (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 $\mathcal{L}$. Finally, we increase t by 1 to let k_t refer to i.

This finishes the rear-processing procedure for updating $\mathcal{L}$. The proof of Lemma 4 is omitted.

▶ **Lemma 4.** *In the case $l_i < p_{i-1} + d_{\min} \leq r_i$, all algorithm invariants hold after I_i is processed.*

2.3.3 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 $\mathcal{L}$ from the front to the rear).

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

Otherwise, we check whether the following is true

$$\frac{l_{k_{s+1}} - l_{k_s}}{k_{s+1} - k_s} > \frac{r_i - l_{k_s}}{i - k_s}. \tag{4}$$

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 $\mathcal{L}$ 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) is true. The front-processing procedure stops if either $s = t$ (i.e., $\mathcal{L}$ only has one element) or Inequality (4) 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 $\mathcal{L}$ 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. The proof of Lemma 5 is omitted.

▶ **Lemma 5.** *In the case $p_{i-1} + d_{\min} > r_i$, all algorithm invariants hold after I_i is processed.*

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. The pseudocode is given in Algorithm 1.

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 $1 \leq i \leq n$

$p_1 \leftarrow l_1$, $i^* \leftarrow 1$, $j^* \leftarrow 1$, $d_{\min} \leftarrow \infty$, $\mathcal{L} \leftarrow \{1\}$;
for $i \leftarrow 2$ **to** n **do**
  if $p_{i-1} + d_{\min} \leq l_i$ **then**
    $p_i \leftarrow l_i$, $\mathcal{L} \leftarrow \{i\}$;
  else
    if $l_i < p_{i-1} + d_{\min} \leq r_i$ **then**
      $p_i \leftarrow p_{i-1} + d_{\min}$;
    else /* $p_{i-1} + d_{\min} > r_i$ */
      $p_i \leftarrow r_i$, $k_s \leftarrow$ the front element of $\mathcal{L}$;
      while $|\mathcal{L}| > 1$ **do** /* the front-processing procedure */
        if $\frac{l_{k_{s+1}} - l_{k_s}}{k_{s+1} - k_s} > \frac{r_i - l_{k_s}}{i - k_s}$ **then**
          for $j \leftarrow k_s + 1$ **to** k_{s+1} **do**
            $p_j \leftarrow l_{k_s} + \frac{l_{k_{s+1}} - l_{k_s}}{k_{s+1} - k_s} \cdot (j - k_s)$;
          remove k_s from $\mathcal{L}$, $k_s \leftarrow$ the front element of $\mathcal{L}$;
        else
          break;
      $i^* \leftarrow k_s$, $j^* \leftarrow i$, $d_{\min} \leftarrow \frac{r_{j^*} - l_{i^*}}{j^* - i^*}$;
    while $|\mathcal{L}| > 1$ **do** /* the rear-processing procedure */
      $k_t \leftarrow$ the rear element of $\mathcal{L}$;
      if $\frac{l_{k_t} - l_{k_{t-1}}}{k_t - k_{t-1}} > \frac{l_i - l_{k_{t-1}}}{i - k_{t-1}}$ **then** break;
      ;
      remove k_t from $\mathcal{L}$;
    add i to the rear of $\mathcal{L}$;
$k_s \leftarrow$ the front element of $\mathcal{L}$;
if $k_s < n$ **then**
  for $j \leftarrow k_s + 1$ **to** n **do**
    $p_j \leftarrow l_{k_s} + d_{\min} \cdot (j - k_s)$;

2.4 The Correctness and the Time Analysis

Based on the algorithm invariants and Corollary 2, the following lemma proves the correctness of the algorithm.

▶ **Lemma 6.** *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 $\mathcal{L} = \{k_s, k_{s+1}, \ldots, k_t\}$ 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 $\mathcal{L}$, 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 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 7.

▶ **Theorem 7.** *Our algorithm computes an optimal solution of the line version of points dispersion problem in $O(n)$ time.*

Proof. By Lemma 6, 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 $\mathcal{L}$. An easy observation is that any element can be removed from $\mathcal{L}$ at most once in the entire algorithm. Hence, the total time of all front-processing procedures in the entire algorithm is $O(n)$.

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 $\mathcal{L}$. Again, since any element can be removed from $\mathcal{L}$ 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. ◀

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., $\min_{1 \le i < j \le n} |p_i p_j|$, 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 $\mathcal{C}$.

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 8.** *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 $|p'_jp'_k| \geq |p'_{j+n}p'_{k+n}|$. 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, $|p'_kp'_{k+n}| \leq n \cdot d$. Since $p'_k = l'_k$ and $p'_{n+k} > l'_{n+k}$, we have $|p'_kp'_{n+k}| > |l'_kl'_{n+k}| = |\mathcal{C}|$. 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 8), and $|l'_kl'_{n+k}| = |\mathcal{C}|$, 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 9.** $d_{\min} = d_{opt}$.

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 8), 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, $|\overrightarrow{p_kp_{k+1}}| \geq d_{opt}$ 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 $d_{opt} \leq d_{\min}$, $d_{opt} = d_{\min}$ holds. 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 10.** *The cycle version of the points dispersion problem is solvable in $O(n)$ time.*

Acknowledgment. The authors would like to thank Minghui Jiang for suggesting the problem to them.

References

1 C. Baur and S. P. Fekete. Approximation of geometric dispersion problems. *Algorithmica*, 30(3):451–470, 2001.

2 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.*, 19(3):267–288, 2009.

3 M. Chrobak, C. Dürr, W. Jawor, L. Kowalik, and M. Kurowski. A note on scheduling equal-length jobs to maximize throughput. *Journal of Scheduling*, 9(1):71–73, 2006.

4 M. Chrobak, W. Jawor, J. Sgall, and T. Tichý. Online scheduling of equal-length jobs: Randomization and restarts help. *SIAM Journal of Computing*, 36(6):1709–1728, 2007.

5 E. Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46:48–60, 1990.

6 E. Fernández, J. Kalcsics, and S. Nickel. The maximum dispersion problem. *Omega*, 41(4):721–730, 2013.

7 R. J. Fowler, M. S. Paterson, and S. L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12:133–137, 1981.

8 Z. Füredi. The densest packing of equal circles into a parallel strip. *Discrete and Computational Geometry*, 6:95–106, 1991.

9 M. R. Garey, D. S. Johnson, B. B. Simons, and R. E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal of Computing*, 10:256–269, 1981.

10 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*, pages 1–10, 2007.

11 T. Lang and E. B. Fernández. Scheduling of unit-length independent tasks with execution constraints. *Information Processing Letters*, 4:95–98, 1976.

12 C. D. Maranasa, C. A. Floudas, and P. M. Pardalosb. New results in the packing of equal circles in a square. *Discrete Mathematics*, 142:287–293, 1995.

13 O. A. Prokopyev, N. Kong, and D. L. Martinez-Torres. The equitable dispersion problem. *European Journal of Operational Research*, 197(1):59–67, 2009.

14 S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Facility dispersion problems: Heuristics and special cases. *Algorithms and Data Structures*, 519:355–366, 1991.

15 S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.

16 B. Simons. A fast algorithm for single processor scheduling. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 246–252, 1978. doi:10.1109/SFCS.1978.4.

17 N. Vakhania. A study of single-machine scheduling problem to maximize throughput. *Journal of Scheduling*, 16(4):395–403, 2013.

18 N. Vakhania and F. Werner. Minimizing maximum lateness of jobs with naturally bounded job data on a single machine in polynomial time. *Theor. Comp. Science*, 501:72–81, 2013.

19 D. W. Wang and Y.-S. Kuo. A study on two geometric location problems. *Information Processing Letters*, 28:281–286, 1988.