On Line-Separable Weighted Unit-Disk Coverage and Related Problems

Gang Liu
Kahlert School of Computing, University of Utah, Salt Lake City, UT, USA

Haitao Wang
Kahlert School of Computing, University of Utah, Salt Lake City, UT, USA

Abstract

Given a set P of n points and a set S of n weighted disks in the plane, the disk coverage problem is to compute a subset of disks of smallest total weight such that the union of the disks in the subset covers all points of P. The problem is NP-hard. In this paper, we consider a line-separable unit-disk version of the problem where all disks have the same radius and their centers are separated from the points of P by a line ℓ. We present an $O(n^{3/2}\log^2 n)$ time algorithm for the problem. This improves the previously best work of $O(n^2\log n)$ time. Our result leads to an algorithm of $O(n^{7/2}\log^2 n)$ time for the halfplane coverage problem (i.e., using n weighted halfplanes to cover n points), an improvement over the previous $O(n^4\log n)$ time solution. If all halfplanes are lower ones, our algorithm runs in $O(n^{3/2}\log^2 n)$ time, while the previous best algorithm takes $O(n^2\log n)$ time. Using duality, the hitting set problems under the same settings can be solved with similar time complexities.

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

Keywords and phrases Line-separable, unit disks, halfplanes, geometric coverage, geometric hitting set

Digital Object Identifier 10.4230/LIPIcs.MFCS.2024.70

Related Version *Full Version*: http://arxiv.org/abs/2407.00329

Funding This research was supported in part by NSF under Grant CCF-2300356.

1 Introduction

Let P be a set of points, and S a set of disks in the plane such that each disk has a positive weight. The *disk coverage* problem asks for a subset of disks whose union covers all points and the total weight of the disks in the subset is minimized. The problem is NP-hard, even if all disks have the same radius and all disks have the same weight [12, 20]. Polynomial-time approximation algorithms have been proposed for the problem and many of its variants, e.g., [1, 7–9, 13, 17].

In this paper, we consider a *line-separable unit-disk* version of the problem where all disks have the same radius and their centers are separated from the points of P by a line ℓ (see Fig. 1). This version of the problem has been studied before. For the *unweighted case*, that is, all disks have the same weight, Ambühl, Erlebach, Mihalák, and Nunkesser [3] first solved the problem in $O(m^2n)$ time, where $n = |P|$ and $m = |S|$. An improved $O(nm + n\log n)$ time algorithm was later given in [11]. Liu and Wang [19] recently presented an $O((n+m)\log(n+m))$ time algorithm.[1] For the weighted case, Pederson and Wang [22] derived an algorithm of $O(nm\log(m+n))$ time or $O((m+n)\log(m+n)+\kappa\log m)$ time,

[1] The runtime of the algorithm in the conference paper of [19] was $m^{2/3}n^{2/3}2^{O(\log^*(m+n))} + O((n+m)\log(n+m))$, which has been improved to $O((n+m)\log(n+m))$ time in the latest arXiv version.

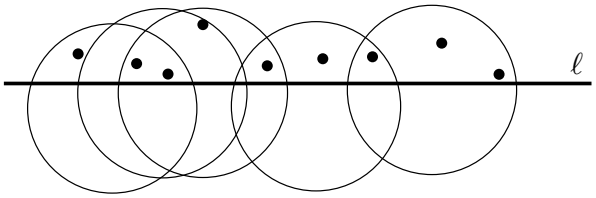

Figure 1 Illustrating the line-separable unit-disk case: All points of P are above ℓ while the centers of all disks are below ℓ.

where κ is the number of pairs of disks that intersect and $\kappa = O(m^2)$ in the worst case. In this paper, we propose an algorithm of $O(n\sqrt{m}\log^2 m + (m+n)\log(m+n))$ time for the weighted case. In addition to the improvement over the previous work, perhaps theoretically more interesting is that the runtime of our algorithm is subquadratic.

The halfplane coverage problem. If every disk of S is a halfplane, then the problem becomes the *halfplane coverage problem*. To solve the problem, Pedersen and Wang [22] showed that the problem can be reduced to $O(n^2)$ instances of the *lower-only halfplane coverage problem* in which all halfplanes are lower halfplanes; this reduction works for both the unweighted and the weighted cases. Consequently, if the lower-only problem can be solved in $O(T)$ time, then the general problem (i.e., S has both lower and upper halfplanes) is solvable in $O(n^2 \cdot T)$ time.

For the weighted lower-only problem, Chan and Grant [8] first gave an algorithm that runs in $O((m+n)^4)$ time. As observed in [22], the lower-only halfplane coverage problem is actually a special case of the line-separable unit-disk coverage problem. Indeed, let ℓ be a horizontal line below all the points of P. Then, since each halfplane of S is a lower halfplane, it can be considered a disk of infinite radius with center below ℓ. In this way, the lower-only halfplane coverage problem becomes an instance of the line-separable unit-disk coverage problem. As such, with their algorithm for the weighted line-separable unit-disk coverage problem, Pederson and Wang [22] solved the weighted lower-only halfplane coverage problem in $O(nm + n\log n)$ time or $O((m+n)\log(m+n) + m^2\log m)$ time. Using our new algorithm for the weighted line-separable unit-disk coverage problem, the weighted lower-only halfplane coverage problem can now be solved in $O(n\sqrt{m}\log^2 m + (m+n)\log(m+n))$ time.

The unweighted lower-only halfplane coverage problem can be solved faster. Indeed, since the problem is a special case of the unweighted line-separable unit-disk coverage problem, with the $O((n+m)\log(n+m))$ time algorithm of Liu and Wang [19] for the latter problem, the unweighted lower-only halfplane coverage problem is solvable in $O((n+m)\log(n+m))$ time. Wang and Xue [24] derived another $O((n+m)\log(n+m))$ time algorithm for the unweighted lower-only halfplane coverage problem with a different approach (which does not work for the unit-disk problem); they also solved the general unweighted halfplane coverage problem in $O(n^{4/3}\log^{5/3} n \log^{O(1)}\log n)$ time. In addition, by a reduction from the set equality problem [4], a lower bound of $\Omega((n+m)\log(n+m))$ is proved in [24] for the lower-only halfplane coverage problem under the algebraic decision tree model.

The hitting set problem. A related problem is the hitting set problem in which each point of P has a positive weight and we seek to find a subset of points with minimum total weight so that each disk of S contains at least one point of the subset. Since the disks of S have the same radius, the problem is actually "dual" to the disk coverage problem. More specifically, if we consider the set of unit disks centered at the points of P as a set of "dual disks" and

consider the centers of the disks of S as a set of "dual points", then the hitting set problem on P and S is equivalent to finding a minimum weight subset of dual disks whose union covers all dual points. Consequently, applying our new weighted case line-separable unit-disk coverage algorithm in this paper solves the weighted line-separable unit-disk hitting set problem in $O(m\sqrt{n}\log^2 n + (m+n)\log(m+n))$ time; applying the $O((m+n)\log(m+n))$ time algorithm in [19] for the unweighted line-separable unit-disk coverage algorithm solves the unweighted line-separable unit-disk hitting set problem in $O((m+n)\log(m+n))$ time.

If every disk of S is a halfplane, then the problem becomes the *halfplane hitting set* problem. Har-Peled and Lee [14] first solved the weighted problem in $O((m+n)^6)$ time. Liu and Wang [18] showed that the problem can be reduced to $O(n^2)$ instances of the *lower-only halfplane hitting set problem* in which all halfplanes are lower halfplanes; this reduction works for both the unweighted and the weighted cases. Consequently, if the lower-only problem can be solved in $O(T)$ time, then the general problem can be solved in $O(n^2 \cdot T)$ time. For the lower-only problem, as in the coverage problem, it is a special case of the line-separable unit-disk hitting set problem; consequently, the weighted and unweighted cases can be solved in $O(m\sqrt{n}\log^2 n + (m+n)\log(m+n))$ time using our new algorithm in this paper and $O((m+n)\log(m+n))$ time using the algorithm in [19], respectively.

Other related work. Pedersen and Wang [22] actually considered a *line-constrained* disk coverage problem, where disk centers are on the x -axis while the points of P can be anywhere in the plane, but the disks may have different radii. They solved the weighted case in $O(nm + n\log n)$ time or $O((m+n)\log(m+n) + \kappa\log m)$ time, where κ is the number of pairs of disks that intersect. For the unweighted case, Liu and Wang [19] gave an algorithm of $O((n+m)\log(m+n) + m\log m\log n)$ time. The line-constrained disk hitting set problem was also studied by Liu and Wang [18], where an $O((m+n)\log(m+n) + \kappa\log m)$ time algorithm was derived for the weighted case, matching the time complexity of the above line-constrained disk coverage problem. Other types of line-constrained problems have also been considered in the literature, e.g., [2, 5, 6, 15, 16, 21, 25].

Our approach. Our algorithm for the weighted line-separable unit-disk coverage problem is essentially a dynamic program. The algorithm description is quite simple and elegant. However, it is not straightforward to prove its correctness. To this end, we show that our algorithm is consistent with the algorithm in [22] for the same problem; one may view our algorithm as a different implementation of the algorithm in [22]. Another challenge of our approach lies in its implementation. More specifically, our algorithm has two key operations, and the efficiency of the algorithm hinges on how to perform these operations efficiently. For this, we construct a data structure based on building a cutting on the disks of S [10]. Although we do not have a good upper bound on the runtime of each individual operation of the algorithm, we manage to bound the total time of all operations in the algorithm by $O(n\sqrt{m}\log^2 m + (m+n)\log(m+n))$.

Outlines. The rest of the paper is structured as follows. After introducing some notation in Section 2, we describe our algorithm and prove its correctness in Section 3. The implementation of the algorithm is presented in Section 4.

2 Preliminaries

We follow the notation defined in Section 1, e.g., P, S, m, n, ℓ. All disks of S have the same radius, which we call *unit disks*. Without loss of generality, we assume that ℓ is the x-axis and all points of P are above ℓ while all centers of disks of S are below ℓ. Note that when we say that a point is above (or below) ℓ, we allow the case where the point is on ℓ.

We assume that each point of P is covered by at least one disk since otherwise there would be no solution. Our algorithm can check whether this assumption is met. For ease of discussion, we make a general position assumption that no point of P lies on the boundary of a disk and no two points of P have the same x-coordinate.

We call a subset S' of S a *feasible subset* if the union of all disks of S' covers all points of P. If S' is a feasible subset of minimum total weight, then S' is called an *optimal subset*. Let δ_{opt} denote the total weight of all disks in an optimal subset; we call δ_{opt} the *optimal objective value*.

For any point q in the plane, let $S_q \subseteq S$ denote the subset of disks containing q; define $\overline{S_q} = S \setminus S_q$. For each disk $s \in S$, let $w(s)$ denote its weight.

3 Algorithm description and correctness

We now present our algorithm. As mentioned above, the algorithm description is quite simple. The challenging part is to prove its correctness and implement it efficiently. In the following, we first describe the algorithm in Section 3.1, and then establish its correctness in Section 3.2. The algorithm implementation will be elaborated in Section 4.

3.1 Algorithm description

We first sort the points of P from left to right as $p_1, p_2, \ldots, p_n$. Our algorithm then processes the points of P in this order. For each point $p_i \in P$, the algorithm computes a value δ_i. The algorithm also maintains a value $cost(s)$ for each disk $s \in S$, which is initialized to its weight $w(s)$. The pseudocode of the algorithm is given in Algorithm 1.

Algorithm 1 The primal algorithm.

Input: The points of P are sorted from left to right as $p_1, p_2, \ldots, p_n$
Output: The optimal objective value δ_{opt}

1 $cost(s) \leftarrow w(s)$, for all disks $s \in S$;
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\quad \delta_i \leftarrow \min_{s \in S_{p_i}} cost(s)$; `// FindMinCost Operation`
4 $\quad cost(s) \leftarrow w(s) + \delta_i$ for all disks $s \in \overline{S_{p_i}}$; `// ResetCost Operation`
5 **end**
6 **return** δ_n;

The algorithm is essentially a dynamic program. We prove in Section 3.2 that the value δ_n returned by the algorithm is equal to δ_{opt}, the optimal objective value. To find an optimal subset, one could slightly modify the algorithm following the standard dynamic programming backtracking technique. More specifically, if δ_n is equal to $cost(s)$ for some disk $s \in S_{p_n}$, then s should be reported as a disk in the optimal subset. Suppose that $cost(s)$ is equal to $w(s) + \delta_i$ for some point p_i. Then δ_i is equal to $cost(s')$ for some disk $s' \in S_{p_i}$ and s' should be reported as a disk in the optimal subset. We continue this backtracking process until a disk whose cost is equal to its own weight is reported (in which case all points of P are covered by the reported disks).

For reference purposes, we use FindMinCost to refer to the operation in Line 3 and use ResetCost to refer to the operation in Line 4 of Algorithm 1. The efficiency of the algorithm hinges on how to implement these two *key operations*, which will be discussed in Section 4.

3.2 Correctness of Algorithm 1

We prove that Algorithm 1 is correct, i.e., prove $\delta_n = \delta_{\text{opt}}$. To this end, we show that our algorithm is consistent with the algorithm of Pederson and Wang [22] for the same problem, or alternatively, our algorithm provides a different implementation of their algorithm. Their algorithm first reduces the problem to a 1D interval coverage problem and then solves the interval coverage problem by a dynamic programming algorithm. In the following, we first review their problem reduction in Section 3.2.1 and then explain their dynamic programming algorithm in Section 3.2.2. Finally, we show that our algorithm is essentially an implementation of their dynamic programming algorithm in Section 3.2.3.

3.2.1 Reducing the problem to an interval coverage problem

For convenience, let p_0 (resp., p_{n+1}) be a point to the left (resp., right) all the points of P and is not contained in any disk of S.

Consider a disk $s \in S$. We say that a subsequence $P[i, j]$ of P with $1 \leq i \leq j \leq n$ is a *maximal subsequence covered* by s if all points of $P[i, j]$ are covered by s but neither p_{i-1} nor p_{j+1} is (it is well defined due to p_0 and p_{n+1}). Define $F(s)$ as the set of all maximal subsequences covered by s. It is easy to see that the subsequences of $F(s)$ are pairwise disjoint.

For each point p_i of P, we vertically project it on the x-axis ℓ; let p_i^* denote the projection point. Let P^* denote the set of all such projection points. Due to our general position assumption that no two points of P have the same x-coordinate, all points of P^* are distinct. For any $1 \leq i \leq j \leq n$, we use $P^*[i, j]$ to denote the subsequence $p_i^*, p_{i+1}^*, \ldots, p_j^*$.

Next, we define a set S^* of line segments on ℓ as follows. For each disk $s \in S$ and each subsequence $P[i, j] \in F(s)$, we create a segment for S^*, denoted by $s^*[i, j]$, with the left endpoint at p_i^* and the right endpoint at p_j^*. As such, $s^*[i, j]$ covers all points of $P^*[i, j]$ and does not cover any point of $P^* \setminus P^*[i, j]$. We let the weight of $s^*[i, j]$ be equal to $w(s)$. We say that $s^*[i, j]$ is *defined* by the disk s.

Consider the following *interval coverage* problem (i.e., each segment of S^* can also be considered an interval of ℓ): Find a subset of segments of S^* of minimum total weight such that the union of the segments in the subset covers all points of P^*. Pederson and Wang [22] proved that an optimal solution to this interval coverage problem corresponds to an optimal solution to the original disk coverage problem on P and S. More specifically, suppose that S^*_{opt} is an optimal subset of the interval coverage problem. Then, we can obtain an optimal subset S_{opt} for the disk coverage problem as follows: For each segment $s^*[i, j] \in S^*_{\text{opt}}$, we add the disk that defines $s^*[i, j]$ to S_{opt}. It is proved in [22] that S_{opt} thus obtained is an optimal subset of the disk coverage problem. Note that since a disk of S may define multiple segments of S^*, a potential issue with S_{opt} is that a disk may be included in S_{opt} multiple times (i.e., if multiple segments defined by the disk are in S^*_{opt}); but this is proved impossible [22].

3.2.2 Solving the interval coverage problem

We now explain the dynamic programming algorithm in [22] for the interval coverage problem.

Let p_0^* be the vertical projection of p_0 on ℓ. Note that $p_0^*, p_1^*, \ldots, p_n^*$ are sorted on ℓ from left to right. For each segment $s^* \in S^*$, let $w(s^*)$ denote the weight of s^*.

For each segment $s^* \in S^*$, define f_{s^*} as the index of the rightmost point of $P^* \cup \{p_0^*\}$ strictly to the left of the left endpoint of s^*. Note that f_{s^*} is well defined due to p_0^*.

For each $i \in [1, n]$, define δ_i^* as the minimum total weight of a subset of segments of S^* whose union covers all points of $P^*[1, i]$. The goal of the interval coverage problem is to compute δ_n^*, which is equal to δ_{opt} [22]. For convenience, we let $\delta_0^* = 0$. For each segment $s^* \in S^*$, define $cost(s^*) = w(s^*) + \delta_{f_{s^*}}^*$. One can verify that $\delta_i^* = \min_{s^* \in S_{p_i^*}^*} cost(s^*)$, where $S_{p_i^*}^* \subseteq S^*$ is the subset of segments that cover p_i^*. This is the recursive relation of the dynamic programming algorithm.

Assuming that the indices f_{s^*} for all disks $s^* \in S^*$ have been computed, the algorithm works as follows. We sweep a point q on ℓ from left to right. Initially, q is at p_0^*. During the sweeping, we maintain the subset $S_q^* \subseteq S^*$ of segments that cover q. The algorithm maintains the invariant that the cost of each segment of S_q^* is already known and the values δ_i^* for all points $p_i^* \in P^*$ to the left of q have been computed. An event happens when q encounters an endpoint of a segment of S^* or a point of P^*. If q encounters a point $p_i^* \in P^*$, then we find the segment of S_q^* with the minimum cost and assign the cost value to δ_i^*. If q encounters the left endpoint of a segment s^*, we set $cost(s^*) = w(s^*) + \delta_{f_{s^*}}^*$ and then insert s^* into S_q^*. If q encounters the right endpoint of a segment, we remove the segment from S_q^*. The algorithm finishes once q meets p_n^*, at which event δ_n^* is computed.

Remark. It was shown in [22] that the above dynamic programming algorithm for the interval coverage problem can be implemented in $O((|P^*| + |S^*|) \log(|P^*| + |S^*|))$. While $|P^*| = n$, $|S^*|$ may be relatively large. A straightforward upper bound for $|S^*|$ is $O(nm)$. Pederson and Wang [22] proved another bound $|S^*| = O(n + m + \kappa)$, where κ is the number of pairs of disks that intersect. This leads to their algorithm of $O(nm \log(m + n) + n \log n)$ time or $O((m + n) \log(m + n) + \kappa \log m)$ time for the original disk coverage problem on P and S.

3.2.3 Correctness of our algorithm

Next, we argue that $\delta_n = \delta_n^*$, which will establish the correctness of Algorithm 1 since $\delta_n^* = \delta_{\text{opt}}$. In fact, we will show that $\delta_i = \delta_i^*$ for all $1 \le i \le n$. We prove it by induction.

As the base case, we first argue $\delta_1 = \delta_1^*$. To see this, by definition, $\delta_1 = \min_{s \in S_{p_1}} w(s)$ because $cost(s) = w(s)$ initially for all disks $s \in S$. For δ_1^*, notice that $f_{s^*} = 0$ for every segment $s^* \in S_{p_1^*}^*$. Since $\delta_0^* = 0$, we have $\delta_1^* = \min_{s^* \in S_{p_1^*}^*} w(s^*)$. By definition, a segment $s^* \in S^*$ covers p_1^* only if the disk of S defining s^* covers p_1, and the segment has the same weight as the disk. Therefore, s^* is in $S_{p_1^*}^*$ only if the disk of S defining s^* is in S_{p_1}. On the other hand, if a disk s covers p_1, then s must define exactly one segment in S^* covering p_1^*. Hence, for each disk $s \in S_{p_1}$, it defines exactly one segment in $S_{p_1^*}^*$ with the same weight. This implies that δ_1^* is equal to the minimum weight of all disks of S covering p_1, and therefore, $\delta_1^* = \delta_1$ must hold.

Consider any i with $2 \le i \le n$. Assuming that $\delta_j = \delta_j^*$ for all $1 \le j < i$, we now prove $\delta_i = \delta_i^*$. Recall that $\delta_i = \min_{s \in S_{p_i}} cost(s)$ and $\delta_i^* = \min_{s^* \in S_{p_i^*}^*} cost(s^*)$. As argued above, each disk $s \in S_{p_i}$ defines a segment in $S_{p_i^*}^*$ with the same weight and each segment $s^* \in S_{p_i^*}^*$ is defined by a disk in S_{p_i} with the same weight. Let s^* be the segment of $S_{p_i^*}^*$ defined by a disk $s \in S_{p_i}$. To prove $\delta_i = \delta_i^*$, it suffices to show that $cost(s)$ of s is equal to $cost(s^*)$ of s^*. To see this, first note that $w(s) = w(s^*)$. By definition, $cost(s^*) = w(s^*) + \delta_{f_{s^*}}^*$. For notational convenience, let $j = f_{s^*}$. By definition, all points of $P^*[j + 1, i]$ are covered by the segment s^* but the point p_j^* is not. Therefore, all points of $P[j + 1, i]$ are covered by the

disk s but p_j is not. As such, during the ResetCost operation of the j-th iteration of the for loop in Algorithm 1, $cost(s)$ will be set to $w(s) + \delta_j$; furthermore, $cost(s)$ will not be reset again during the i'-th iteration for all $j + 1 \leq i' \leq i$. Therefore, we have $cost(s) = w(s) + \delta_j$ at the beginning of the i-th iteration of the algorithm. Since $\delta_j = \delta_j^*$ holds by induction hypothesis and $w(s) = w(s^*)$, we obtain $cost(s) = cost(s^*)$. This proves $\delta_i = \delta_i^*$.

The correctness of Algorithm 1 is thus established.

Remark. The above proof for $\delta_n = \delta_{\text{opt}}$ also implies that $\delta_i = \delta_{\text{opt}}^i$ for all $1 \leq i \leq n-1$, where δ_{opt}^i is the minimum total weight of a subset of disks whose union covers all points of $P[1, i]$. Indeed, we can apply the same proof to the points of $P[1, i]$ only. Observe that δ_i will never change after the i-th iteration of Algorithm 1.

4 Algorithm implementation

In this section, we discuss the implementation of Algorithm 1. Specifically, we describe how to implement the two key operations FindMinCost and ResetCost. A straightforward method can implement each operation in $O(m)$ time, resulting in a total $O(mn + n \log n)$ time of the algorithm. Note that this is already a logarithmic factor improvement over the previous work of Pederson and Wang [22]. In the following, we present a faster approach of $O(n\sqrt{m} \log^2 m + (m + n) \log(m + n))$ time.

Duality. Recall that the points of P are sorted from left to right as $p_1, p_2, \ldots, p_n$. In fact, we consider the problem in the "dual" setting. For each point $p_i \in P$, let d_i denote the unit disk centered at p_i, and we call d_i the *dual disk* of p_i. For each disk $s \in S$, let q_s denote the center of s, and we call q_s the *dual point* of s. We define the weight of q_s to be equal to $w(s)$. We use D to denote the set of all dual disks and Q the set of all dual points. For each dual point $q \in Q$, let $w(q)$ denote its weight. Because all disks of S are unit disks, we have the following observation.

▶ **Observation 1.** *A disk $s \in S$ covers a point $p_i \in P$ if and only if the dual point q_s is covered by the dual disk d_i.*

For any disk $d_i \in D$, let Q_{d_i} denote the subset of dual points of Q that are covered by d_i, i.e., $Q_{d_i} = Q \cap d_i$. Define $\overline{Q_{d_i}} = Q \setminus Q_{d_i}$. In light of Observation 1, Algorithm 1 is equivalent to the following Algorithm 2.

Algorithm 2 An algorithm "dual" to Algorithm 1.

```
1 cost(q) ← w(q), for all dual points q ∈ Q;
2 for i ← 1 to n do
3     δ_i ← min_{q∈Q_{d_i}} cost(q);                    // FindMinCost Operation
4     cost(q) ← w(q) + δ_i for all dual points q ∈ Q̄_{d_i};   // ResetCost Operation
5 end
6 return δ_n;
```

In the following, we will present an implementation for Algorithm 2, and in particular, for the two operations FindMinCost and ResetCost.

For each disk $d_i \in D$, since its center is above the x-axis ℓ and all points of Q are below ℓ, only the portion of d_i below ℓ matters to Algorithm 2. We call the boundary portion of d_i below ℓ the *lower arc* of d_i. Let H denote the set of the lower arcs of all disks of D.

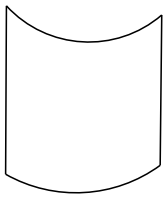

Figure 2 Illustrating a pseudo-trapezoid.

Cuttings. Our algorithm will need to construct a cutting on the arcs of H [10]. We explain this concept first. Note that $|H| = n$. For a parameter r with $1 \leq r \leq n$, a $(1/r)$-*cutting* Ξ of size $O(r^2)$ for H is a collection of $O(r^2)$ constant-complexity cells whose union covers the entire plane and whose interiors are pairwise disjoint such that the interior of each cell $\sigma \in \Xi$ is crossed by at most n/r arcs of H, i.e., $|H_\sigma| \leq n/r$, where H_σ is the subset of arcs of H that cross the interior of σ (H_σ is often called the *conflict list* in the literature). Let D_σ be the subset of disks of D whose lower arcs are in H_σ. Hence, we also have $|D_\sigma| \leq n/r$.

We actually need to construct a *hierarchical cutting* for H [10]. We say that a cutting Ξ' *c-refines* another cutting Ξ if each cell of Ξ' is completely inside a single cell of Ξ and each cell of Ξ contains at most c cells of Ξ'. Let Ξ_0 denote the cutting with a single cell that is the entire plane. We define cuttings $\{\Xi_0, \Xi_1, ..., \Xi_k\}$, in which each Ξ_i, $1 \leq i \leq k$, is a $(1/\rho^i)$-cutting of size $O(\rho^{2i})$ that c-refines Ξ_{i-1}, for two constants ρ and c. By setting $k = \lceil \log_\rho r \rceil$, the last cutting Ξ_k is a $(1/r)$-cutting. The sequence $\{\Xi_0, \Xi_1, ..., \Xi_k\}$ is called a *hierarchical* $(1/r)$-*cutting* for H. If a cell σ' of Ξ_{i-1}, $1 \leq i \leq k$, contains cell σ of Ξ_i, we say that σ' is the *parent* of σ and σ is a *child* of σ'. We can also define *ancestors* and *descendants* correspondingly. As such, the hierarchical $(1/r)$-cutting can be viewed as a tree structure with the single cell of Ξ_0 as the root. We often use Ξ to denote the set of all cells in all cuttings Ξ_i, $0 \leq i \leq k$. The total number of cells of Ξ is $O(r^2)$ [10].

A hierarchical $(1/r)$-cutting of H can be computed in $O(nr)$ time, e.g., by the algorithm in [23], which adapts Chazelle's algorithm [10] for hyperplanes. The algorithm also produces the conflict lists H_σ (and thus D_σ) for all cells $\sigma \in \Xi$, implying that the total size of these conflict lists is bounded by $O(nr)$. In particular, each cell of the cutting produced by the algorithm of [23] is a (possibly unbounded) *pseudo-trapezoid* that typically has two vertical line segments as left and right sides, a sub-arc of an arc of H as a top side (resp., bottom side) (see Fig. 2).

In what follows, we first discuss a preprocessing step in Section 4.1. The algorithms for handling the two key operations are described in the subsequent two subsections, respectively. Section 4.4 finally summarizes everything.

4.1 Preprocessing

In order to handle the two key operations, we first perform some preprocessing work before we run Algorithm 2. As discussed above, we first sort all points of P from left to right. In the following, we describe a data structure which will be used to support the two key operations.

We start by computing a hierarchical $(1/r)$-cutting $\{\Xi_0, \Xi_1, ..., \Xi_k\}$ for H in $O(nr)$ time [10,23], for a parameter $1 \leq r \leq n$ to be determined later. We follow the above notation, e.g., σ, H_σ, D_σ, Ξ. As discussed above, the cutting algorithm also produces the conflict lists H_σ (and thus D_σ) for all cells $\sigma \in \Xi$. Using the conflict lists, we compute a list $L(d_i)$ for each disk $d_i \in D$, where $L(d_i)$ comprises all cells $\sigma \in \Xi$ such that $d_i \in D_\sigma$. Computing $L(d_i)$ for all disks $d_i \in D$ can be done in $O(\sum_{\sigma \in \Xi} |H_\sigma|)$ time by simply traversing the conflict lists H_σ of all cells $\sigma \in \Xi$, which takes $O(nr)$ time as $\sum_{\sigma \in \Xi} |H_\sigma| = O(nr)$. Note that this also implies $\sum_{d_i \in D} |L(d_i)| = O(nr)$.

For any region R in the plane, let $Q(R)$ denote the subset of points of Q that are inside R, i.e., $Q(R) = Q \cap R$.

For simplicity, we assume that no point of Q is on the boundary of any cell of Ξ. This implies that each point of Q is in the interior of a single cell of Ξ_i, for all $0 \leq i \leq k$. We compute the subset $Q(\sigma)$ of all cells σ in the last cutting Ξ_k. This can be done by a point location procedure as follows. For each point $q \in Q$, starting from the only cell of Ξ_0, we find the cell σ_i of Ξ_i containing q, for each $0 \leq i \leq k$. More precisely, suppose that σ_i is known. To find σ_{i+1}, we simply check all $O(1)$ children of σ_i and find the one that contains q, which takes $O(1)$ time. As such, processing all points of Q takes $O(m \log r)$ time as $k = O(\log r)$, after which $Q(\sigma)$ for all cells $\sigma \in \Xi_k$ are computed. We explicitly store $Q(\sigma)$ for all cells $\sigma \in \Xi_k$. Note that the subsets $Q(\sigma)$ for all cells $\sigma \in \Xi_k$ form a partition of Q. Therefore, we have the following observation.

▶ **Observation 2.** $\sum_{\sigma \in \Xi_k} |Q(\sigma)| = m$.

Note that a cell $\sigma \in \Xi$ is the ancestor of another cell $\sigma' \in \Xi$ (alternatively, σ' is a descendant of σ) if and only if σ fully contains σ'. For convenience, we consider σ an ancestor of itself but not a descendant of itself. Let $A(\sigma)$ denote the set of all ancestors of σ and $B(\sigma)$ the set of all descendants of σ. Hence, σ is in $A(\sigma)$ but not in $B(\sigma)$. Let $C(\sigma)$ denote the set of all children of σ Clealy, $|A(\sigma)| = O(\log r)$ and $|C(\sigma)| = O(1)$.

Variables and algorithm invariants. For each point $q \in Q$, we associate with it a variable $\lambda(q)$. For each cell $\sigma \in \Xi$, we associate with it two variables: $minCost(\sigma)$ and $\lambda(\sigma)$. If $|Q(\sigma)| = \emptyset$, then $minCost(\sigma) = \infty$ and $\lambda(\sigma) = 0$ always hold during the algorithm. Our algorithm for handling the two key operations will maintain the following two invariants.

▶ **Algorithm Invariant 1.** *For any point $q \in Q$, $cost(q) = w(q) + \lambda(q) + \sum_{\sigma' \in A(\sigma)} \lambda(\sigma')$, where σ is the cell of Ξ_k that contains q.*

▶ **Algorithm Invariant 2.** *For each cell $\sigma \in \Xi$ with $Q(\sigma) \neq \emptyset$, if σ is a cell of Ξ_k, then $minCost(\sigma) = \min_{q \in Q(\sigma)}(w(q) + \lambda(q))$; otherwise, $minCost(\sigma) = \min_{\sigma' \in C(\sigma)}(minCost(\sigma') + \lambda(\sigma'))$.*

The above algorithm invariants further imply the following observation.

▶ **Observation 3.** *For each cell $\sigma \in \Xi$ with $Q(\sigma) \neq \emptyset$, $\min_{q \in Q(\sigma)} cost(q) = minCost(\sigma) + \sum_{\sigma' \in A(\sigma)} \lambda(\sigma')$.*

Proof. We prove the observation by induction. For the base case, consider a cell σ of the last cutting Ξ_k. By the two algorithm invariants, we have

$$\begin{aligned}
\min_{q \in Q(\sigma)} cost(q) &= \min_{q \in Q(\sigma)} \Big(w(q) + \lambda(q) + \sum_{\sigma' \in A(\sigma)} \lambda(\sigma') \Big) && \text{by Algorithm Invariant 1} \\
&= \min_{q \in Q(\sigma)} \Big(w(q) + \lambda(q) \Big) + \sum_{\sigma' \in A(\sigma)} \lambda(\sigma') \\
&= minCost(\sigma) + \sum_{\sigma' \in A(\sigma)} \lambda(\sigma'). && \text{by Algorithm Invariant 2}
\end{aligned}$$

This proves the observation for σ.

Now consider a cell $\sigma \in \Xi \setminus \Xi_k$. We assume that the observation holds for all children σ' of σ, i.e., $\min_{q \in Q(\sigma')} cost(q) = minCost(\sigma') + \sum_{\sigma'' \in A(\sigma')} \lambda(\sigma'')$. Then, we have

$$\begin{aligned}
&\min_{q \in Q(\sigma)} cost(q) = \min_{\sigma' \in C(\sigma)} \min_{q \in Q(\sigma')} cost(q) \\
&= \min_{\sigma' \in C(\sigma)} \Big(minCost(\sigma') + \sum_{\sigma'' \in A(\sigma')} \lambda(\sigma'') \Big) && \text{by induction hypothesis} \\
&= \min_{\sigma' \in C(\sigma)} \Big(minCost(\sigma') + \lambda(\sigma') \Big) + \sum_{\sigma'' \in A(\sigma)} \lambda(\sigma'') \\
&= minCost(\sigma) + \sum_{\sigma'' \in A(\sigma)} \lambda(\sigma''). && \text{by Algorithm Invariant 2}
\end{aligned}$$

This proves the observation. ◀

For each cell $\sigma \in \Xi$, we also maintain $\mathcal{L}(\sigma)$, a list comprising all descendant cells σ' of σ whose values $\lambda(\sigma')$ are not zero and all points $q \in Q(\sigma)$ whose values $\lambda(q)$ are not zero. As $\mathcal{L}(\sigma)$ has both cells of $B(\sigma)$ and points of $Q(\sigma)$, for convenience, we use an "element" to refer to either a cell or a point of $\mathcal{L}(\sigma)$. As such, for any element $e \in B(\sigma) \cup Q(\sigma)$ with $e \notin \mathcal{L}(\sigma)$, $\lambda(e) = 0$ must hold. As will be seen later, whenever the algorithm sets $\lambda(\sigma)$ to a nonzero value for a cell $\sigma \in \Xi$, σ will be added to $\mathcal{L}(\sigma')$ for every ancestor σ' of σ with $\sigma' \neq \sigma$. Similarly, whenever the algorithm sets $\lambda(q)$ to a nonzero value for a point q, then q will be added to $\mathcal{L}(\sigma')$ for every cell $\sigma' \in A(\sigma)$, where σ is the cell of Ξ_k containing q.

Initialization. The above describes the data structure. We now initialize the data structure, and in particular, initialize the variables $\mathcal{L}(\cdot)$, $\lambda(\cdot)$, $minCost(\cdot)$ so that the algorithm invariants hold.

First of all, for each cell $\sigma \in \Xi$, we set $\mathcal{L}(\sigma) = \emptyset$ and $\lambda(\sigma) = 0$. For each point $q \in Q$, we set $\lambda(q) = 0$. Since $cost(q) = w(q)$ initially, it is not difficult to see that Algorithm Invariant 1 holds.

We next set $minCost(\sigma)$ for all cells of $\sigma \in \Xi$ in a bottom-up manner following the tree structure of Ξ. Specifically, for each cell σ in the last cutting Ξ_k, we set $minCost(\sigma) = \min_{q \in Q(\sigma)} w(q)$ by simply checking every point of $Q(\sigma)$. If $Q(\sigma) = \emptyset$, we set $minCost(\sigma) = \infty$. This establishes the second algorithm invariant for all cells $\sigma \in \Xi_k$. Then, we set $minCost(\sigma)$ for all cells of $\sigma \in \Xi_{k-1}$ with $minCost(\sigma) = \min_{\sigma' \in C(\sigma)}(minCost(\sigma') + \lambda(\sigma'))$, after which the second algorithm invariant holds for all cells $\sigma \in \Xi_{k-1}$. We continue this process to set $minCost(\sigma)$ for cells in $\Xi_{k-2}, \Xi_{k-3}, \ldots, \Xi_0$. After that, the second algorithm invariant is established for all cells $\sigma \in \Xi$.

In addition, for each cell σ in the last cutting Ξ_k, in order to efficiently update $minCost(\sigma)$ once $\lambda(q)$ changes for a point $q \in Q(\sigma)$, we construct a min-heap $\mathcal{H}(\sigma)$ on all points q of $Q(\sigma)$ with the values $w(q) + \lambda(q)$ as "keys". Using the heap, if $\lambda(q)$ changes for a point $q \in Q(\sigma)$, $minCost(\sigma)$ can be updated in $O(\log m)$ time as $|Q(\sigma)| \leq m$.

This finishes our preprocessing step for Algorithm 2. The following lemma analyzes the time complexity of the preprocessing.

▶ **Lemma 4.** *The preprocessing takes $O(n \log n + nr + m \log r)$ time.*

Proof. First of all, sorting P takes $O(n \log n)$ time. Constructing the hierarchical cutting Ξ takes $O(nr)$ time. Computing the lists $L(d_i)$ for all disks $d_i \in D$ also takes $O(nr)$ time. The point location procedure for computing the subsets $Q(\sigma)$ for all cells $\sigma \in \Xi_k$ runs in $O(m \log r)$ time. For the initialization step, setting $\mathcal{L}(Q) = \emptyset$ and $\lambda(\sigma) = 0$ for all cells $\sigma \in \Xi$

takes $O(r^2)$ time as $|\Xi| = O(r^2)$. Since Ξ_k has $O(r^2)$ cells, computing $minCost(\sigma)$ for all cells $\sigma \in \Xi_k$ can be done in $O(r^2 + \sum_{\sigma \in \Xi_k} |Q(\sigma)|)$ time, which is $O(r^2 + m)$ by Observation 2. Initializing $minCost(\sigma)$ for all other cells $\sigma \in \Xi \setminus \Xi_k$ takes $O(r^2)$ time since each cell has $O(1)$ children (and thus computing $minCost(\sigma)$ for each such cell σ takes $O(1)$ time). Finally, constructing a heap $\mathcal{H}(\sigma)$ for all cells $\sigma \in \Xi_k$ takes $O(\sum_{\sigma \in \Xi_k} |Q(\sigma)|)$ time, which is $O(m)$ by Observation 2. Since $r \leq n$, $r^2 \leq nr$. Therefore, the total time of the preprocessing is $O(n \log n + nr + m \log r)$. ◀

4.2 The FindMinCost operation

We now discuss how to perform the FindMinCost operation.

Consider a disk d_i in FindMinCost operation of the i-th iteration of Algorithm 2. The goal is to compute $\min_{q \in Q_{d_i}} cost(q)$, i.e., the minimum cost of all points of Q inside the disk d_i.

Recall that $L(d_i)$ is the list of all cells $\sigma \in \Xi$ such that $d_i \in D_\sigma$. Define $L_1(d_i)$ to be the set of all cells of $L(d_i)$ that are from Ξ_k and let $L_2(d_i) = L(d_i) \setminus L_1(d_i)$. Define $L_3(d_i)$ as the set of cells $\sigma \in \Xi$ such that σ's parent is in $L_2(d_i)$ and σ is completely contained in d_i. We first have the following observation following the definition of the hierarchical cutting.

▶ **Observation 5.** *Q_{d_i} is the union of $\bigcup_{\sigma \in L_1(d_i)} (Q(\sigma) \cap d_i)$ and $\bigcup_{\sigma \in L_3(d_i)} Q(\sigma)$.*

Proof. Consider a point $q \in \bigcup_{\sigma \in L_1(d_i)} (Q(\sigma) \cap d_i)$. Suppose that q is in $Q(\sigma) \cap d_i$ for some cell $\sigma \in L_1(d_i)$. Then, since $q \in d_i$, it is obvious true that $q \in Q_{d_i}$.

Consider a point $q \in \bigcup_{\sigma \in L_3(d_i)} Q(\sigma)$. Suppose that $q \in Q(\sigma)$ for some cell $\sigma \in L_3(d_i)$. By the definition of $L_3(d)$, σ is fully contained in d_i. Therefore, $q \in d_i$ holds. Hence, $q \in Q_{d_i}$.

On the other hand, consider a point $q \in Q_{d_i}$. By definition, d_i contains q. Let σ be the cell of Ξ_k containing q. Since both d_i and σ contain q, $d_i \cap \sigma \neq \emptyset$. Therefore, either $\sigma \subseteq d_i$ or the boundary of d_i crosses σ. In the latter case, we have $\sigma \in L_1(d_i)$ and thus $q \in \bigcup_{\sigma \in L_1(d_i)} (Q(\sigma) \cap d_i)$. In the former case, σ must have two ancestors σ_1 and σ_2 such that (1) σ_1 is the parent of σ_2; (2) σ_2 is fully contained in d_i; (3) the boundary of d_i crosses σ_1. This is true because σ is fully contained in d_i while the boundary of d_i crosses the only cell of Ξ_0, which is the entire plane and is an ancestor of σ. As such, σ_1 must be in $L_2(d_i)$ and σ_2 must be in $L_3(d_i)$. Therefore, q must be in $\bigcup_{\sigma \in L_3(d_i)} Q(\sigma)$.

This proves the observation. ◀

With Observation 5, we now describe our algorithm for FindMinCost. Let α be a variable, which is initialized to ∞. At the end of the algorithm, we will have $\alpha = \min_{q \in Q_{d_i}} cost(q)$. For each cell σ in the list $L(d_i)$, if it is from $\sigma \in \Xi_k$, i.e., $\sigma \in L_1(d_i)$, then we process σ as follows. For each point $q \in Q(\sigma)$, by Algorithm Invariant 1, we have $cost(q) = w(q) + \lambda(q) + \sum_{\sigma' \in A(\sigma)} \lambda(\sigma')$. If $q \in d_i$, we compute $cost(q)$ by visiting all cells of $A(\sigma)$, which takes $O(\log r)$ time, and then we update $\alpha = \min\{\alpha, cost(q)\}$.

If $\sigma \in L_2(d_i)$, then we process it as follows. For each child σ' of σ that is fully contained in d_i (i.e., $\sigma \in L_3(d_i)$), we compute $\alpha_{\sigma'} = minCost(\sigma') + \sum_{\sigma'' \in A(\sigma')} \lambda(\sigma'')$ by visiting all cells of $A(\sigma')$, which takes $O(\log r)$ time. By Observation 3, we have $\alpha_{\sigma'} = \min_{q \in Q(\sigma)} cost(q)$. Then we update $\alpha = \min\{\alpha, \alpha_{\sigma'}\}$. After processing every cell $\sigma \in L(d_i)$ as above, we return α, which is equal to $\min_{q \in Q_{d_i}} cost(q)$ according to our algorithm invariants as well as Observation 5. This finishes the FindMinCost operation. The following lemma analyzes the runtime of the operation.

▶ **Lemma 6.** *The total time of the FindMinCost operations in the entire Algorithm 2 is bounded by $O((nr + mn/r) \log r)$.*

Proof. Recall that in each operation we processes cells of $L_1(d_i)$ and cells of $L_2(d_i)$ in different ways. The total time of the operation is the sum of the time for processing $L_1(d_i)$ and the time for processing $L_2(d_i)$.

For the time for processing $L_1(d_i)$, for each cell $\sigma \in L_1(d_i)$, for each point $q \in Q(\sigma) \cap d_i$, we spend $O(\log r)$ time computing $cost(q)$. Hence, the time for processing cells of $L_1(d_i)$ for each d_i is bounded by $O(\sum_{\sigma \in L_1(d_i)} |Q(\sigma)| \log r)$. The total time of processing $L_1(d_i)$ in the entire algorithm is on the order of $\sum_{i=1}^{n} \sum_{\sigma \in L_1(d_i)} |Q(\sigma)| \cdot \log r = \sum_{\sigma \in \Xi_k} (|D_\sigma| \cdot |Q(\sigma)|) \cdot \log r$. Recall that $|D_\sigma| \leq n/r$ for each cell $\sigma \in \Xi_k$. Hence, $\sum_{\sigma \in \Xi_k} (|D_\sigma| \cdot |Q(\sigma)|) \leq n/r \cdot \sum_{\sigma \in \Xi_k} |Q(\sigma)|$, which is $O(mn/r)$ by Observation 2. Therefore, the total time for processing cells of $L_1(d_i)$ in the entire Algorithm 2 is $O(mn/r \cdot \log r)$.

For the time for processing $L_2(d_i)$, for each cell $\sigma \in L_2(d_i)$, for each child σ' of σ, it takes $O(\log r)$ time to compute $\alpha_{\sigma'}$. Since σ has $O(1)$ cells, the total time for processing all cells of $L_2(d_i)$ is $O(|L_2(d_i)| \cdot \log r)$. The total time of processing $L_2(d_i)$ in the entire algorithm is on the order of $\sum_{i=1}^{n} |L_2(d_i)| \cdot \log r$. Note that $\sum_{i=1}^{n} |L_2(d_i)| \leq \sum_{i=1}^{n} |L(d_i)| = O(nr)$. Therefore, the total time for processing cells of $L_2(d_i)$ in the entire Algorithm 2 is $O(nr \log r)$.

Summing up the time for processing $L_1(d_i)$ and $L_2(d_i)$ leads to the lemma. ◀

4.3 The ResetCost operation

We now discuss the ResetCost operation. Consider the ResetCost operation in the i-th iteration of Algorithm 2. The goal is to reset $cost(q) = w(q) + \delta_i$ for all points $q \in Q$ that are outside the disk d_i. To this end, we will update our data structure, and more specifically, update the $\lambda(\cdot)$ and $minCost(\cdot)$ values for certain cells of Ξ and points of Q so that the algorithm invariants still hold.

Define $L_4(d_i)$ as the set of cells $\sigma \in \Xi$ such that σ's parent is in $L_2(d_i)$ and σ is completely outside d_i. Let $\overline{d_i}$ denote the region of the plane outside the disk d_i. We have the following observation, which is analogous to Observation 5.

▶ **Observation 7.** *$\overline{Q_{d_i}}$ is the union of $\bigcup_{\sigma \in L_1(d_i)} (Q(\sigma) \cap \overline{d_i})$ and $\bigcup_{\sigma \in L_4(d_i)} Q(\sigma)$.*

Proof. The proof is the same as that of Observation 5 except that we use $\overline{d_i}$ to replace d_i and use $L_4(d_i)$ to replace $L_3(d_i)$. We omit the details. ◀

Our algorithm for ResetCost works as follows. Consider a cell $\sigma \in L(d_i)$. As for the FindMinCost operation, depending on whether σ is from $L_1(d_i)$ or $L_2(d_i)$, we process it in different ways.

If σ is from $L_1(d_i)$, we process σ as follows. For each point $q \in Q(\sigma)$, if $q \in \overline{d_j}$, then we are supposed to reset $cost(q)$ to $w(q) + \delta_i$. To achieve the effect and also maintain the algorithm invariants, we do the following. First, we set $\lambda(q) = \delta_i - \sum_{\sigma' \in A(\sigma)} \lambda(\sigma')$, which can be done in $O(\log r)$ time by visiting the ancestors of σ. As such, we have $w(q) + \lambda(q) + \sum_{\sigma' \in A(\sigma)} \lambda(\sigma') = w(q) + \delta_i$, which establishes the first algorithm invariant for q. For the second algorithm invariant, we first update $minCost(\sigma)$ using the heap $\mathcal{H}(\sigma)$, i.e., by updating the key of q to the new value $w(q) + \lambda(q)$. The heap operation takes $O(\log m)$ time. Next, we update $minCost(\sigma')$ for all ancestors σ' of σ in a bottom-up manner using the formula $minCost(\sigma') = \min_{\sigma'' \in C(\sigma')} (minCost(\sigma'') + \lambda(\sigma''))$. Since each cell has $O(1)$ children, updating all ancestors of σ takes $O(\log r)$ time. This establishes the second algorithm invariant. Finally, since $\lambda(q)$ has just been changed, if $\lambda(q) \neq 0$, then we add q to the list $\mathcal{L}(\sigma')$ for all cells $\sigma' \in A(\sigma)$. Note that for each such $\mathcal{L}(\sigma')$ it is possible that q was already in the list before; but we do not check this and simply add q to the end of the list (and thus the list may contain multiple copies of q). This finishes the processing of q, which takes $O(\log r + \log m)$ time. Processing all points of $q \in Q(\sigma)$ as above takes $O(|Q(\sigma)| \cdot (\log r + \log m))$ time.

If σ is from $L_2(d_i)$, then we process σ as follows. For each child σ' of σ, if σ' is completely outside d_i, then we process σ' as follows. We are supposed to reset $cost(q)$ to $w(q) + \delta_i$ for all points $q \in Q(\sigma')$. In other words, the first algorithm invariant does not hold any more and we need to update our data structure to restore it. Note that the second algorithm invariant still holds. To achieve the effect and also maintain the algorithm invariants, we do the following. For each element e in the list $\mathcal{L}(\sigma')$ (recall that e is either a cell of $B(\sigma')$ or a point of $Q(\sigma')$), we process e as follows. First, we remove e from $\mathcal{L}(\sigma')$. Then we reset $\lambda(e) = 0$. If e is a point of $Q(\sigma')$, then let σ_e be the cell of Ξ_k that contains e; otherwise, e is a cell of $B(\sigma')$ and let σ_e be the parent of e. Since $\lambda(e)$ is changed, we update $minCost(\sigma'')$ for all cells $\sigma'' \in A(\sigma_e)$ in the same way as above in the first case for processing $L_1(d_i)$, which takes $O(\log r + \log m)$ time. This finishes processing e, after which the second algorithm invariant still holds. After all elements of $\mathcal{L}(\sigma')$ are processed as above, $\mathcal{L}(\sigma')$ becomes $\emptyset$ and we reset $\lambda(\sigma') = \delta_i - \sum_{\sigma'' \in A(\sigma') \setminus \{\sigma'\}} \lambda(\sigma'')$. Since $\lambda(\sigma')$ has been changed, we update $minCost(\sigma'')$ for all cells $\sigma'' \in A(\sigma)$ in the same way as before (which takes $O(\log r)$ time), after which the second algorithm invariant still holds. In addition, if $\lambda(\sigma') \neq 0$, then we add σ' to the list $\mathcal{L}(\sigma'')$ for all cells $\sigma'' \in A(\sigma)$, which again takes $O(\log r)$ time. This finishes processing σ', which takes $O(|\mathcal{L}(\sigma')| \cdot (\log r + \log m))$ time. It remains to restore the first algorithm invariant, for which we have the following observation.

▶ **Observation 8.** *After σ' is processed, the first algorithm invariant is established for all points $q \in Q(\sigma')$.*

Proof. Consider a point $q \in Q(\sigma')$. It suffices to show $w(q) + \delta_i = w(q) + \lambda(q) + \sum_{\sigma'' \in A(\sigma_q)} \lambda(\sigma'')$, where σ_q is the cell of Ξ_k that contains q. After the elements of the list $\mathcal{L}(\sigma')$ are processed as above, we have $\lambda(q) = 0$ for all points $q \in Q(\sigma')$ and $\lambda(\sigma'') = 0$ for all descendants σ'' of σ'. Therefore, $w(q) + \lambda(q) + \sum_{\sigma'' \in A(\sigma_1)} \lambda(\sigma_1) = w(q) + \sum_{\sigma'' \in A(\sigma')} \lambda(\sigma'') = w(q) + \lambda(\sigma') + \sum_{\sigma'' \in A(\sigma') \setminus \{\sigma'\}} \lambda(\sigma'')$. Recall that $\lambda(\sigma') = \delta_i - \sum_{\sigma'' \in A(\sigma') \setminus \{\sigma'\}} \lambda(\sigma'')$. We thus obtain $w(q) + \lambda(q) + \sum_{\sigma'' \in A(\sigma_q)} \lambda(\sigma'') = w(q) + \delta_i$. ◀

This finishes the ResetCost operation. According to Observation 7, $cost(q)$ has been reset for all points $q \in Q$ that are outside d_i. The following lemma analyzes the runtime of the operation.

▶ **Lemma 9.** *The total time of the ResetCost operations in the entire Algorithm 2 is bounded by $O((nr + mn/r) \cdot \log r \cdot (\log r + \log m))$.*

Proof. Recall that we processes cells of $L_1(d_i)$ and cells of $L_2(d_i)$ in different ways. The total time of the operation is the sum of the time for processing $L_1(d_i)$ and the time for processing $L_2(d_i)$.

For the time for processing $L_1(d_i)$, for each cell $\sigma \in L_1(d_i)$, recall that processing all points of $Q(\sigma)$ takes $O(|Q(\sigma)| \cdot (\log r + \log m))$ time. Hence, the total time for processing cells of $L_1(d_i)$ is on the order of $\sum_{\sigma \in L_1(d_i)} |Q(\sigma)| \cdot (\log r + \log m)$. The total time for processing cells of $L_1(d_i)$ in the entire Algorithm 2 is thus on the order of $\sum_{i=1}^{n} \sum_{\sigma \in L_1(d_i)} |Q(\sigma)| \cdot (\log r + \log m)$. As analyzed in the proof of Lemma 6, $\sum_{i=1}^{n} \sum_{\sigma \in L_1(d_i)} |Q(\sigma)| = O(mn/r)$. Therefore, the total time for processing cells of $L_1(d_i)$ in the entire Algorithm 2 is $O(mn/r \cdot (\log r + \log m))$.

For the time for processing $L_2(d_i)$, for each cell $\sigma \in L_2(d_i)$, for each child σ' of σ, processing σ' takes $O(|\mathcal{L}(\sigma')| \cdot (\log r + \log m))$ time. Next, we give an upper bound for $|\mathcal{L}(\sigma')|$ for all such cells σ' in the entire algorithm. Recall that each element e of $\mathcal{L}(\sigma')$ is either a point $q \in Q(\sigma')$ or a descendant cell $\sigma'' \in B(\sigma')$. Let $\mathcal{L}_1(\sigma')$ denote the subset of elements of $\mathcal{L}(\sigma')$ in the former case and $\mathcal{L}_2(\sigma')$ the subset of elements in the latter case. In the following we provide an upper bound for each subset.

1. For $\mathcal{L}_1(\sigma')$, notice that a point q is in the list only if σ_q is crossed by the lower arc of a disk d_i, where σ_q is the cell of Ξ_k containing q. If q is outside d_i, then a copy of q will be added to $\mathcal{L}(\sigma'')$ for all $O(\log r)$ cells σ'' of $A(\sigma_q)$. As $|D_{\sigma_q}| \leq n/r$, the number of elements in $\mathcal{L}_1(\sigma')$ contributed by the points of $Q(\sigma_q)$ for all such cells σ' in the entire algorithm is bounded by $O(|Q(\sigma_q)| \cdot n/r \cdot \log r)$. In light of Observation 2, the total size of $\mathcal{L}_1(\sigma')$ of all such cells σ' in the entire Algorithm 2 is $O(mn/r \cdot \log r)$.
2. For $\mathcal{L}_2(\sigma')$, observe that a cell σ_1 is in the list only if σ_2 is crossed by the lower arc of a disk d_i, where σ_2 is the parent of σ_1. If σ_1 is completely outside d_i, then a copy of σ_1 is added to $\mathcal{L}(\sigma'')$ for all $O(\log r)$ cells σ'' of $A(\sigma_2)$. As such, the number of elements in $\mathcal{L}_2(\sigma')$ for all such cells σ' in the entire algorithm contributed by each cell $\sigma_1 \in \Xi$ is bounded by $O(|D_{\sigma_2}| \cdot \log r)$. Since every cell of Ξ has $O(1)$ children and $\sum_{\sigma_2 \in \Xi} |D_{\sigma_2}| = O(nr)$, the total size of $\mathcal{L}_2(\sigma')$ of all such cells σ' in the entire Algorithm 2 is $O(nr \cdot \log r)$.

Therefore, the total time for processing cells of $L_2(d_i)$ in the entire Algorithm 2 is $O((nr + mn/r) \cdot \log r \cdot (\log r + \log m))$.

Summing up the time for processing $L_1(d_i)$ and $L_2(d_i)$ leads to the lemma. ◀

4.4 Putting it all together

We summarize the time complexity of the overall algorithm. By Lemma 4, the preprocessing step takes $O(n \log n + nr + m \log r)$ time. By Lemma 6, the total time for performing the FindMinCost operations in the entire algorithm is $O((nr + mn/r) \cdot \log r)$. By Lemma 9, the total time for performing the ResetCost operations in the entire algorithm is $O((nr + mn/r) \cdot \log r \cdot (\log m + \log r))$. Therefore, the total time of the overall algorithm is $O(n \log n + m \log r + (nr + mn/r) \cdot \log r \cdot (\log m + \log r))$. Recall that $1 \leq r \leq n$. Setting $r = \min\{\sqrt{m}, n\}$ gives the upper bound $O(n\sqrt{m} \log^2 m + (n + m) \log(n + m))$ for the time complexity of the overall algorithm.

Note that we have assumed that each point of P is covered by at least one disk of S. In this is not the case, then no feasible subset exists (alternatively, one may consider the optimal objective value ∞); if we run our algorithm in this case, then one can check that the value δ_n returned by our algorithm is ∞. Hence, our algorithm can automatically determine whether a feasible subset exists.[2]

▶ **Theorem 10.** *Given a set of n points and a set of m weighted unit disks in the plane such that the points and the disk centers are separated by a line, there is an $O(n\sqrt{m} \log^2 m + (n + m) \log(n + m))$ time algorithm to compute a subset of disks of minimum total weight whose union covers all points.*

References

1 Pankaj K. Agarwal and Jiangwei Pan. Near-linear algorithms for geometric hitting sets and set covers. *Discrete and Computational Geometry*, 63:460–482, 2020. doi:10.1007/s00454-019-00099-6.

2 Helmut Alt, Esther M. Arkin, Hervé Brönnimann, Jeff Erickson, Sándor P. Fekete, Christian Knauer, Jonathan Lenchner, Joseph S. B. Mitchell, and Kim Whittlesey. Minimum-cost coverage of point sets by disks. In *Proceedings of the 22nd Annual Symposium on Computational Geometry (SoCG)*, pages 449–458, 2006. doi:10.1145/1137856.1137922.

[2] It is possible to determine whether a feasible subset exists in $O(n \log n)$ time. For example, one can first compute the upper envelope of the boundary portions of all disks above ℓ. Then, it suffices to determine whether every point of P is below the upper envelope. Note that the upper envelope is x-monotone.

3 Christoph Ambühl, Thomas Erlebach, Matúš Mihalák, and Marc Nunkesser. Constant-factor approximation for minimum-weight (connected) dominating sets in unit disk graphs. In *Proceedings of the 9th International Conference on Approximation Algorithms for Combinatorial Optimization Problems (APPROX), and the 10th International Conference on Randomization and Computation (RANDOM)*, pages 3–14, 2006. doi:10.1007/11830924_3.

4 Michael Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 80–86, 1983. doi:10.1145/800061.808735.

5 Vittorio Bilò, Ioannis Caragiannis, Christos Kaklamanis, and Panagiotis Kanellopoulos. Geometric clustering to minimize the sum of cluster sizes. In *Proceedings of the 13th European Symposium on Algorithms (ESA)*, pages 460–471, 2005. doi:10.1007/11561071_42.

6 Ahmad Biniaz, Prosenjit Bose, Paz Carmi, Anil Maheshwari, J. Ian Munro, and Michiel Smid. Faster algorithms for some optimization problems on collinear points. In *Proceedings of the 34th International Symposium on Computational Geometry (SoCG)*, pages 8:1–8:14, 2018. doi:10.4230/LIPIcs.SoCG.2018.8.

7 Norbert Bus, Nabil H. Mustafa, and Saurabh Ray. Practical and efficient algorithms for the geometric hitting set problem. *Discrete Applied Mathematics*, 240:25–32, 2018. doi:10.1016/j.dam.2017.12.018.

8 Timothy M. Chan and Elyot Grant. Exact algorithms and APX-hardness results for geometric packing and covering problems. *Computational Geometry: Theory and Applications*, 47:112–124, 2014. doi:10.1016/j.comgeo.2012.04.001.

9 Timothy M. Chan and Qizheng He. Faster approximation algorithms for geometric set cover. In *Proceedings of 36th International Symposium on Computational Geometry (SoCG)*, pages 27:1–27:14, 2020. doi:10.4230/LIPIcs.SoCG.2020.27.

10 Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete & Computational Geometry*, 9:145–158, 1993. doi:10.1007/BF02189314.

11 Francisco Claude, Gautam K. Das, Reza Dorrigiv, Stephane Durocher, Robert Fraser, Alejandro López-Ortiz, Bradford G. Nickerson, and Alejandro Salinger. An improved line-separable algorithm for discrete unit disk cover. *Discrete Mathematics, Algorithms and Applications*, 2:77–88, 2010. doi:10.1142/S1793830910000486.

12 Tomás Feder and Daniel H. Greene. Optimal algorithms for approximate clustering. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 434–444, 1988. doi:10.1145/62212.62255.

13 Shashidhara K. Ganjugunte. *Geometric hitting sets and their variants*. PhD thesis, Duke University, 2011. URL: https://dukespace.lib.duke.edu/items/391e2278-74f0-408f-9be7-4c97cf72e352.

14 Sariel Har-Peled and Mira Lee. Weighted geometric set cover problems revisited. *Journal of Computational Geometry*, 3:65–85, 2012. doi:10.20382/jocg.v3i1a4.

15 Arindam Karmakar, Sandip Das, Subhas C. Nandy, and Binay K. Bhattacharya. Some variations on constrained minimum enclosing circle problem. *Journal of Combinatorial Optimization*, 25(2):176–190, 2013. doi:10.1007/s10878-012-9452-4.

16 Nissan Lev-Tov and David Peleg. Polynomial time approximation schemes for base station coverage with minimum total radii. *Computer Networks*, 47:489–501, 2005. doi:10.1016/j.comnet.2004.08.012.

17 Jian Li and Yifei Jin. A PTAS for the weighted unit disk cover problem. In *Proceedings of the 42nd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 898–909, 2015. doi:10.1007/978-3-662-47672-7_73.

18 Gang Liu and Haitao Wang. Geometric hitting set for line-constrained disks. In *Proceedings of the 18th Algorithms and Data Structures Symposium (WADS)*, pages 574–587, 2023. doi:10.1007/978-3-031-38906-1_38.

19 Gang Liu and Haitao Wang. On the line-separable unit-disk coverage and related problems. In *Proceedings of the 34th International Symposium on Algorithms and Computation (ISAAC)*, pages 51:1–51:14, 2023. Full version available at https://arxiv.org/abs/2309.03162.

20 Nabil H. Mustafa and Saurabh Ray. Improved results on geometric hitting set problems. *Discrete and Computational Geometry*, 44:883–895, 2010. doi:10.1007/s00454-010-9285-9.

21 Logan Pedersen and Haitao Wang. On the coverage of points in the plane by disks centered at a line. In *Proceedings of the 30th Canadian Conference on Computational Geometry (CCCG)*, pages 158–164, 2018. URL: https://home.cs.umanitoba.ca/~cccg2018/papers/session4A-p1.pdf.

22 Logan Pedersen and Haitao Wang. Algorithms for the line-constrained disk coverage and related problems. *Computational Geometry: Theory and Applications*, 105-106:101883:1–18, 2022. doi:10.1016/j.comgeo.2022.101883.

23 Haitao Wang. Unit-disk range searching and applications. *Journal of Computational Geometry*, 14:343–394, 2023. doi:10.20382/jocg.v14i1a13.

24 Haitao Wang and Jie Xue. Algorithms for halfplane coverage and related problems. In *Proceedings of the 40th International Symposium on Computational Geometry (SoCG)*, pages 79:1–79:15, 2024. doi:10.4230/LIPIcs.SoCG.2024.79.

25 Haitao Wang and Jingru Zhang. Line-constrained k-median, k-means, and k-center problems in the plane. *International Journal of Computational Geometry and Application*, 26:185–210, 2016. doi:10.1142/S0218195916600049.