Algorithms for the Line-Constrained Disk Coverage and Related Problems

Logan Pedersen and Haitao Wang(✉)

Department of Computer Science, Utah State University, Logan, UT 84322, USA
logan.pedersen@aggiemail.usu.edu, haitao.wang@usu.edu

Abstract. Given a set P of n points and a set S of m weighted disks in the plane, the disk coverage problem asks for a subset of disks of minimum total weight that cover all points of P. The problem is NP-hard. In this paper, we consider a line-constrained version in which all disks are centered on a line L (while points of P can be anywhere in the plane). We present an $O((m+n)\log(m+n)+\kappa\log m)$ time algorithm for the problem, where κ is the number of pairs of disks that intersect. For the unit-disk case where all disks have the same radius, the running time can be reduced to $O((n+m)\log(m+n))$. In addition, we solve in $O((m+n)\log(m+n))$ time the L_∞ and L_1 cases of the problem, in which the disks are squares and diamonds, respectively. Using our techniques, we further solve two other geometric coverage problems. Given in the plane a set P of n points and a set S of n weighted half-planes, we solve in $O(n^4\log n)$ time the problem of finding a subset of half-planes to cover P so that their total weight is minimized. This improves the previous best algorithm of $O(n^5)$ time by almost a linear factor. If all half-planes are lower ones, our algorithm runs in $O(n^2\log n)$ time, which improves the previous best algorithm of $O(n^4)$ time by almost a quadratic factor.

Keywords: Disk coverage · Line-constrained · Half-plane coverage · Geometric coverage · Facility location

1 Introduction

Given a set P of n points and a set S of m disks in the plane such that each disk has a weight, the *disk coverage* problem asks for a subset of disks of minimum total weight that cover all points of P. We assume that the union of all disks covers all points of P. The problem is known to be NP-hard [11] and approximation algorithms have been proposed, e.g., [17,19].

In this paper, we consider a line-constrained version of the problem in which all disks (possibly with different radii) have their centers on a line L, say, the x-axis. To the best of our knowledge, this line-constrained problem was not particularly studied before. We present an $O((m+n)\log(m+n)+\kappa\log m)$ time

This research was supported in part by NSF under Grant CCF-2005323. A full version of this paper is available at https://arxiv.org/abs/2104.14680.

A. Lubiw and M. Salavatipour (Eds.): WADS 2021, LNCS 12808, pp. 585–598, 2021.
https://doi.org/10.1007/978-3-030-83508-8_42

algorithm, where κ is the number of pairs of disks that intersect (and thus $\kappa \leq m(m-1)/2$; e.g., if the disks are disjoint, then $\kappa = 0$ and the algorithm runs in $O((m+n)\log(m+n))$ time). For the *unit-disk case* where all disks have the same radius, the running time can be reduced to $O((n+m)\log(m+n))$. We also solve in $O((m+n)\log(m+n))$ time the L_∞ and L_1 cases of the problem, in which the disks are squares and diamonds, respectively. As a by-product, we obtain an $O((m+n)\log(m+n))$ time algorithm for the 1D version of the problem where all points of P are on L and the disks are line segments of L. In addition, we show that the problem has an $\Omega((m+n)\log(m+n))$ time lower bound in the algebraic decision tree model even for the 1D case. This implies that our algorithms for the 1D, L_∞, L_1, and unit-disk cases are all optimal.

Our algorithms potentially have applications, e.g., in facility locations. For example, suppose we want to build some facilities along a railway which is represented by L (although an entire railway may not be a straight line, it may be considered straight in a local region) to provide service for some customers that are represented by the points of P. The center of a disk represents a candidate location for building a facility that can serve the customers covered by the disk and the cost for building the facility is the weight of the disk. The problem is to determine the best locations to build facilities so that all customers can be served and the total cost is minimized. This is exactly an instance of our problem.

Although the problems are line-constrained, our techniques can actually be used to solve other geometric coverage problems. If all disks of S have the same radius and the set of disk centers are separated from P by a line ℓ, the problem is called *line-separable unit-disk coverage*. The unweighted case of the problem where the weights of all disks are 1 has been studied in the literature [2,9,10]. In particular, the fastest algorithm was given by Claude et al. [9] and the runtime is $O(n \log n + nm)$. The algorithm, however, does not work for the weighted case. Our algorithm for the line-constrained L_2 case can be used to solve the weighted case in $O(nm\log(m+n))$ time or in $O((m+n)\log(m+n) + \kappa \log m)$ time, where κ is the number of pairs of disks that intersect on the side of ℓ that contains P. More interestingly, we can use the algorithm to solve the following *half-plane coverage problem*. Given in the plane a set P of n points and a set S of m weighted half-planes, find a subset of the half-planes to cover all points of P so that their total weight is minimized. For the *lower-only case* where all half-planes are lower ones, Chan and Grant [8] gave an $O(mn^2(m+n))$ time algorithm. In light of the observation that a half-plane is a special disk of infinite radius, our line-separable unit-disk coverage algorithm can be applied to solve the problem in $O(nm\log(m+n))$ time or in $O(n\log n + m^2\log m)$ time. This improves the result of [8] by almost a quadratic factor (note that the techniques of [8] are applicable to more general problem settings such as downward shadows of x-monotone curves). For the general case where both upper and lower half-planes are present, Har-Peled and Lee [13] proposed an algorithm of $O(n^5)$ time when $m = n$. By using our lower-only case algorithm, we solve the problem in $O(n^3 m\log(m+n))$ time or in $O(n^3\log n + n^2m^2\log m)$ time. Hence, our result improves the one in [13] by almost a linear factor.

1.1 Related Work

Our problem is a new type of set cover. The general set cover problem, which is fundamental and has been studied extensively, is hard to solve, even approximately [12,14,18]. Many set cover problems in geometric settings, often called geometric coverage problems, are also NP-hard, e.g., [8,13]. As mentioned above, if the line-constrained condition is dropped, then the disk coverage problem becomes NP-hard, even if all disks are unit disks with the same weight [11]. Polynomial time approximation schemes (PTAS) exist for the unweighted problem [19] as well as the weighted unit-disk case [17].

Alt et al. [1] studied a problem closely related to ours, with the same input, consisting of P, S, and L, and the objective is also to find a subset of disks of minimum total weight that cover all points of P. But the difference is that S is comprised of all possible disks centered at L and the weight of each disk is defined as r^α with r being the radius of the disk and α being a given constant at least 1. Alt et al. [1] gave an $O(n^4 \log n)$ time algorithm for any L_p metric and any $\alpha \geq 1$, an $O(n^2 \log n)$ time algorithm for any L_p metric and $\alpha = 1$, and an $O(n^3 \log n)$ time algorithm for the L_∞ metric and any $\alpha \geq 1$. Recently, Pedersen and Wang [20] improved all these results by providing an $O(n^2)$ time algorithm for any L_p metric and any $\alpha \geq 1$. A 1D variation of the problem was studied in the literature where points of P are all on L and another set Q of m points is given on L as the only candidate centers for disks. Bilò et al. [5] first showed that the problem is solvable in polynomial time. Lev-Tov and Peleg [16] gave an algorithm of $O((n+m)^3)$ time for any $\alpha \geq 1$. Biniaz et al. [6] recently proposed an $O((n+m)^2)$ time algorithm for the case $\alpha = 1$. Pedersen and Wang [20] solved the problem in $O(n(n+m) + m \log m)$ time for any $\alpha \geq 1$.

Other line-constrained problems have also been studied in the literature, e.g., [15,21].

1.2 Our Approach

We first solve the 1D problem by a simple dynamic programming algorithm. Then, for the "1.5D" problem (i.e., points of P are in the plane), an observation is that if the points of P are sorted by their x-coordinates, then the sorted list can be partitioned into sublists such that there exists an optimal solution in which each disk covers a sublist. Based on the observation, we reduce the 1.5D problem to an instance of the 1D problem with a set P' of n points and a set S' of segments. But two challenges arise.

The first challenge is to give a small bound on $|S'|$. A naive method shows that $|S'| \leq n \cdot m$. In the unit-disk case and the L_1 case, we prove that $|S'|$ can be reduced to m by similar methods. In the L_∞ case, we show that $|S'|$ can be bounded by $2(n+m)$. The most challenging case is the L_2 case. By a number of observations, we prove that $|S'| \leq 2(n+m) + \kappa$.

The second challenge is to compute the set S' (P', which actually consists of all projections of the points of P onto L, can be easily obtained in $O(n)$ time). Our algorithms for computing S' for all cases use the sweeping technique. The algorithms for the unit-disk case and the L_1 case are relatively easy, while those for the L_∞ and L_2 cases require much more effort. Although the two algorithms for L_∞ and L_2 are similar in spirit, the intersections of the disks in the L_2 case bring more difficulties and make the algorithm more involved and less efficient. In summary, computing S' can be done in $O((n+m)\log(n+m))$ time for all cases except the L_2 case which takes $O((n+m)\log(n+m)+\kappa\log m)$ time.

Outline. The rest of the paper is organized as follows. We define notation in Sect. 2. The algorithms for the L_∞ and L_2 cases are given in Sect. 3. Due to the space limit, lemma proofs, algorithms for the unit-disk, and L_1 cases, the lower bound proof (which is based on a reduction from the element uniqueness problem), algorithms for the line-separable disk coverage and half-plane coverage problems are all omitted but can be found in the full paper.

2 Preliminaries

We assume that L is the x-axis. We also assume that all points of P are above or on L because if a point p_i is below L, then we could obtain the same optimal solution by replacing p_i with its symmetric point with respect to L. For ease of exposition, we make a general position assumption that no two points of P have the same x-coordinate and no point of P lies on the boundary of a disk of S.

For any point p in the plane, we use $x(p)$ and $y(p)$ to refer to its x-coordinate and y-coordinate, respectively. We sort all points of P by their x-coordinates, and let $p_1, p_2, \ldots, p_n$ be the sorted list from left to right on L. For any $1 \leq i \leq j \leq n$, let $P[i,j]$ denote the subset $\{p_i, p_{i+1}, \ldots, p_j\}$. Sometimes we use indices to refer to points of P, e.g., point i refers to p_i.

We sort all disks of S by the x-coordinates of their centers from left to right, and let $s_1, s_2, \ldots, s_m$ be the sorted list. For each s_i, let c_i denote its center and w_i denote its weight. We assume that each w_i is positive (otherwise one could always include s_i in the solution). For each disk s_i, let l_i and r_i refer to its leftmost and rightmost points, respectively.

We often talk about the relative positions of two geometric objects O_1 and O_2 (e.g., two points, or a point and a line). We say that O_1 is to the *left* of O_2 if $x(p) \leq x(p')$ holds for any point $p \in O_1$ and any point $p' \in O_2$, and *strictly left* means $x(p) < x(p')$. Similarly, we can define *right, above, below*, etc.

For convenience, we use p_0 (resp., p_{n+1}) to denote a point on L strictly to the left (resp. right) of all points of P and all disks of S. We use the term *optimal solution subset* to refer to a subset of S used in an optimal solution.

In the 1D problem, each disk $s_i \in S$ is a line segment on L. The problem can be solved by a straightforward dynamic programming algorithm of $O((n+m)\log(n+m))$ time. The details are omitted but can be found in the full paper.

3 The L_∞ and L_2 Cases

In this section, we give our algorithms for the L_∞ and L_2 cases. The algorithms are similar in the high level. However, the L_2 case is more involved in the low level computations. In Sect. 3.1, we present a high-level algorithmic scheme that works for both metrics. Then, we complete the algorithms for the L_∞ and L_2 cases in Sects. 3.2 and 3.3, respectively.

3.1 An Algorithmic Scheme for L_∞ and L_2 Metrics

In this subsection, unless otherwise stated, all statements are applicable to both metrics. Note that a disk in the L_∞ metric is a square.

For a disk $s_k \in S$, 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_k if all points of $P[i,j]$ are covered by s_k but neither p_{i-1} nor p_{j+1} is covered by s_k (it is well defined due to p_0 and p_{n+1}). Let $F(s_k)$ be the set of all maximal subsequences covered by s_k. Note that the subsequences of $F(s_k)$ are pairwise disjoint.

Lemma 1. *Suppose S_{opt} is an optimal solution subset and s_k is a disk of S_{opt}. Then, there is a subsequence $P[i,j]$ in $F(s_k)$ such that the following hold.*

1. *$P[i,j]$ has a point that is not covered by any disk in $S_{opt} \setminus \{s_k\}$.*
2. *For any point $p \in P$ that is covered by s_k but is not in $P[i,j]$, p is covered by a disk in $S_{opt} \setminus \{s_k\}$.*

In light of Lemma 1, we reduce the problem to an instance of the 1D problem with a point set P' and a line segment set S', as follows.

For each point of P, we vertically project it on L, and the set P' is comprised of all such projected points. Thus P' has exactly n points. For any $1 \leq i \leq j \leq n$, we use $P'[i,j]$ to denote the projections of the points of $P[i,j]$. For each point $p_i \in P$, we use p'_i to denote its projection point in P'.

The set S' is defined as follows. For each disk $s_k \in S$ and each subsequence $P[i,j] \in F(s_k)$, we create a segment for S', denoted by $s[i,j]$, with left endpoint at p'_i and right endpoint at p'_j. Thus, $s[i,j]$ covers exactly the points of $P'[i,j]$. We set the weight of $s[i,j]$ to w_k. Note that if $s[i,j]$ is already in S', which is defined by another disk s_h, then we only need to update its weight to w_k in case $w_k < w_h$ (so each segment appears only once in S'). We say that $s[i,j]$ is defined by s_k (resp., s_h) if its weight is equal to w_k (resp., w_h).

By Lemma 1, we intend to say that an optimal solution OPT' to the 1D problem on P' and S' corresponds to an optimal solution OPT to the original problem on P and S as follows: if a segment $s[i,j] \in S'$ is included in OPT', then we include the disk that defines $s[i,j]$ in OPT. However, since a disk of S may define multiple segments of S', to guarantee the correctness of the correspondence, we need to show that OPT' is a *valid solution*: no two segments in OPT' are defined by the same disk of S. For this, we have the following lemma.

Lemma 2. *Any optimal solution on P' and S' is a valid solution.*

With our algorithm for the 1D problem, we have the following result.

Lemma 3. *If the set S' is computed, then an optimal solution can be found in $O((n + |S'|)\log(n + |S'|))$ time.*

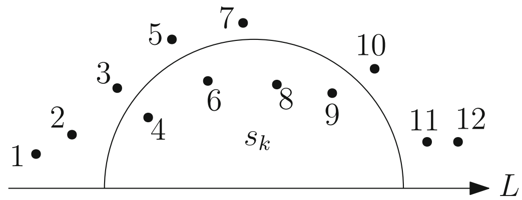

Fig. 1. Illustrating the definition of bounding couples: the numbers are the indices of the points of P. In this example, $p_l(s_k)$ is point 2 and $p_r(s_k)$ is point 11, and the bounding couples are: $(2, 3)$, $(3, 5)$, $(5, 7)$, $(7, 10)$, $(10, 11)$.

It remains to determine the size of S' and compute S'. An obvious answer is that $|S'|$ is bounded by $m \cdot \lceil n/2 \rceil$ because each disk can have at most $\lceil n/2 \rceil$ maximal sequences of P, and a trivial algorithm can compute S' in $O(nm \log(m+n))$ time by scanning the sorted list P for each disk. Therefore, by Lemma 3, we can solve the problem in both L_∞ and L_2 metrics in $O(nm \log(m + n))$ time.

With more geometric observations, we will prove the following two lemmas.

Lemma 4. *In the L_∞ metric, $|S'| \leq 2(n + m)$ and S' can be computed in $O((n + m) \log(n + m))$ time.*

Lemma 5. *In the L_2 metric, $|S'| \leq 2(n + m) + \kappa$ and S' can be computed in $O((n + m) \log(n + m) + \kappa \log m)$ time, where κ is the number of pairs of disks of S that intersect each other.*

With Lemma 3, we can solve the L_∞ case in $O((n+m) \log(n+m))$ time and the L_2 case in $O((n + m) \log(n + m) + \kappa \log m)$ time.

Bounding Couples. Before moving on, we introduce a new concept *bounding couples*, which will be used to prove Lemmas 4 and 5 later.

Consider a disk $s_k \in S$. Let $p_l(s_k)$ denote the rightmost point of $P \cup \{p_0, p_{n+1}\}$ strictly to the left of l_k; similarly, let $p_r(s_k)$ denote the leftmost point of $P \cup \{p_0, p_{n+1}\}$ strictly to the right of r_k. Let $P(s_k)$ denote the subset of points of P between $p_l(s_k)$ and $p_r(s_k)$ inclusively that are outside s_k. We sort the points of $P(s_k)$ by their x-coordinates, and we call each adjacent pair of points (or their indices) in the sorted list a *bounding couple* (e.g., see Fig. 1). Let $C(s_k)$ denote the set of all bounding couples of s_k, and for each bounding couple of $C(s_k)$, we assign w_k to it as the weight. Let $\mathcal{C} = \bigcup_{1 \leq k \leq m} C(s_k)$, and if the same bounding couple is defined by multiple disks, we only keep the copy in $\mathcal{C}$ with the minimum weight. Also, we consider a bounding couple (i, j) as an ordered pair with $i < j$, and i is considered as the left end of the couple while j is the right end.

The reason why we define bounding couples is that if $P[i, j]$ is a maximal subsequence of P covered by s_k then $(i - 1, j + 1)$ is a bounding couple. On the other hand, if (i, j) is a bounding couple of $C(s_k)$, then $P[i+1, j-1]$ is a maximal subsequence of P covered by s_k unless $j = i + 1$. Hence, each bounding couple (i, j) of $\mathcal{C}$ with $j \neq i + 1$ corresponds to a segment in the set S', and $|S'| \leq |\mathcal{C}|$.

Observe that $\mathcal{C}$ has at most $n-1$ couples (i,j) with $j=i+1$, and given $\mathcal{C}$, we can obtain S' in additional $O(|\mathcal{C}|)$ time. According to our above discussion, to prove Lemmas 4 and 5, it suffices to prove the following two lemmas.

Lemma 6. *In the L_∞ metric, $|\mathcal{C}| \le 2(n+m)$ and $\mathcal{C}$ can be computed in $O((n+m)\log(n+m))$ time.*

Lemma 7. *In the L_2 metric, $|\mathcal{C}| \le 2(n+m)+\kappa$ and $\mathcal{C}$ can be computed in $O((n+m)\log(n+m)+\kappa\log m)$ time.*

Consider a bounding couple (i,j) of $\mathcal{C}$, defined by a disk s_k. We call it a *left bounding couple* if $p_i = p_l(s_k)$, a *right bounding couple* if $p_j = p_r(s_k)$, and a *middle bounding couple* otherwise (e.g., in Fig. 1, $(2,3)$ is the left bounding couple, $(10,11)$ is the right bounding couple, and the rest are middle bounding couples). Note that a disk can define at most one left bounding couple and at most one right bounding couple. Therefore, the number of left and right bounding couples in $\mathcal{C}$ is at most $2m$. It remains to bound the number of middle bounding couples of $\mathcal{C}$. We will prove Lemma 6 and 7 in Sects. 3.2 and 3.3, respectively.

3.2 The L_∞ Metric

In this section, our goal is to prove Lemma 6. In the L_∞ metric, every disk is a square that has four axis-parallel edges. We use l_k and r_k to particularly refer to the left and right endpoints of the upper edge of s_k, respectively.

For a point p_i and a square s_k, we say that p_i is *vertically above (resp., below)* the upper edge of s_k if p_i is above (resp., below) the upper edge of s_k and $x(l_k) \le x(p_i) \le x(r_k)$. Due to our general position assumption, p_i is not on the boundary of s_k, and thus p_i above/below the upper edge of s_k implies that p_i is strictly above/below the edge. Also, since no point of P is below L, a point $p_i \in P$ is in s_k if and only if p_i is vertically below the upper edge of s_k. If p_i is vertically above the upper edge of s_k, we also say that p_i is vertically above s_k or s_k is vertically below p_i. The following lemma proves an upper bound for $|\mathcal{C}|$.

Lemma 8. $|\mathcal{C}| \le 2(n+m)$.

We proceed to compute the set $\mathcal{C}$. The following lemma gives an algorithm to compute all left and right bounding couples of $\mathcal{C}$.

Lemma 9. *All left and right bounding couples of $\mathcal{C}$ can be computed in $O((n+m)\log(n+m))$ time.*

Computing the Middle Bounding Couples We now compute all middle bounding couples of $\mathcal{C}$. We sweep a vertical line l from left to right, and an event happens if l encounters a point in $P \cup \{l_k, r_k | \ 1 \le k \le m\}$. Let H be the set of disks that intersect l. During the sweeping, we maintain the following information and invariants (e.g., see Fig. 2).

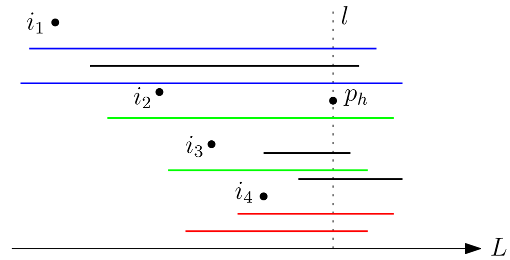

Fig. 2. In this example, $P(l) = \{p_{i_1}, p_{i_2}, p_{i_3}, p_{i_4}\}$. Each horizontal segment represents the upper edge of a disk. $H(i_1)$ consists of two blue disks and $H(i_4)$ consists of two red disks. H_0 consists of three black disks. After processing the event at p_h, i_2, i_3, and i_4 will be removed from $P(l)$ and p_h will be inserted, so after the event $P(l) = \{p_{i_1}, p_h\}$. (i_2, h), (i_3, h), (i_4, h) will be reported as middle bounding couples. (Color figure online)

1. A sequence $P(l) = \{p_{i_1}, p_{i_2}, \ldots, p_{i_t}\}$ of t points of P, which are to the left of l and ordered from northwest to southeast. $P(l)$ is stored in a balanced binary search tree $T(P(l))$.
2. A collection $\mathcal{H}$ of $t+1$ subsets of H: $H(i_j)$ for $j = 0, 1, \ldots, t$, which form a partition of H, defined as follows.
   $H(i_t)$ is the subset of disks of H that are vertically below p_{i_t}. For each $j = t-1, t-2, \ldots, 1$, $H(i_j)$ is the subset of disks of $H \setminus \bigcup_{k=j+1}^{t} H(i_k)$ that are vertically below p_{i_j}. $H(i_0) = H \setminus \bigcup_{j=1}^{t} H(i_j)$. While $H(i_0)$ may be empty, none of $H(i_j)$ for $1 \leq j \leq t$ is empty.
   Each $H(i_j)$ is maintained by a balanced binary search tree $T(H(i_j))$ ordered by the y-coordinates of the upper edges of the disks. We have all disks stored in leaves of $T(H(i_j))$, and each internal node v of the tree also stores a weight equal to the minimum weight of all disks in the leaves of the subtree at v.
3. For each point $p_{i_j} \in P(l)$, among all points of P strictly between p_{i_j} and l, no point is vertically above any disk of $H(i_j)$.
4. Among all points of P strictly to the left of l, no point is vertically above any disk of $H(i_0)$.

In summary, our algorithm maintains the following trees: $T(P(l))$, $T(H(i_j))$ for all $j \in [0, t]$. Initially when l is to the left of all disks and points of P, we have $H = \emptyset$ and $P(l) = \emptyset$. We next describe how to process events.

If l encounters the left endpoint l_k of a disk s_k, we insert s_k to $H(i_0)$. The time for processing this event is $O(\log m)$ since $|H(i_0)| \leq m$.

If l encounters the right endpoint r_k of a disk s_k, we need to determine which set $H(i_j)$ of $\mathcal{H}$ contains s_k. For this, we associate each right endpoint with its disk in the preprocessing so that it can keep track of which set of $\mathcal{H}$ contains the disk. Using this mechanism, we can determine the set $H(i_j)$ that contains s_k in constant time. We then remove s_k from $T(H(i_j))$. If $H(i_j)$ becomes empty and $j \neq 0$, then we remove p_{i_j} from $P(l)$. One can verify that all algorithm invariants still hold. The time for processing this event is $O(\log(m+n))$.

If l encounters a point p_h of P, which is a major event we need to handle, we process it as follows. We search $T(P(l))$ to find the first point p_{i_j} of $P(l)$ below p_h (e.g., $j = 3$ in Fig. 2). We remove the points p_{i_k} for all $k \in [j, t]$ from $P(l)$.

Lemma 10. *For each point p_{i_k} with $k \in [j, t]$, (i_k, h) is a middle bounding couple defined by and only by the disks of $H(i_k)$ (i.e., $H(i_k)$ consists of all disks of S that define (i_k, h) as a middle bounding couple).*

By Lemma 10, for each $k \in [j, t]$, we report (i_k, h) as a middle bounding couple with weight equal to the minimum weight of all disks of $H(i_k)$, which is stored at the root of $T(H(i_k))$.

Next, we process the point $p_{i_{j-1}}$, for which we have the following lemma. The proof technique is similar to that for Lemma 10, so we omit it.

Lemma 11. *If p_h is vertically below the lowest disk of $H(i_{j-1})$, then (i_{j-1}, h) is not a middle bounding couple; otherwise, (i_{j-1}, h) is a middle bounding couple defined by and only by disks of H_{j-1} that are vertically below p_h.*

By Lemma 11, we first check whether p_h is vertically below the lowest disk of $H(i_{j-1})$. If yes, we do nothing. Otherwise, we report (i_{j-1}, h) as a middle bounding couple with weight equal to the minimum weight of all disks of $H(i_{j-1})$ vertically below p_h, which can be computed in $O(\log m)$ time by using weights at the internal nodes of $T(H(i_{j-1}))$. We further have the following lemma.

Lemma 12. *If all disks of $H(i_{j-1})$ are vertically below p_h, then there does not exist a middle bounding couple (i_{j-1}, b) with $b > h$.*

We check whether p_h is above the highest disk of $H(i_{j-1})$ using the tree $T(H(i_{j-1}))$. If yes, then the above lemma tells that there will be no more middle bounding couples involving i_{j-1} any more, and thus we remove $p_{i_{j-1}}$ from $P(l)$.

The following lemma implies that all middle bounding couples with p_h as the right end have been computed.

Lemma 13. *For any middle bounding couple (b, h), b must be in $\{i_{j-1}, i_j, \ldots, i_t\}$.*

Next, we add p_h to the end of the current sequence $P(l)$ (note that the points p_{i_k} for all $k \in [j, t]$ and possibly $p_{i_{j-1}}$ have been removed from $P(l)$; e.g., see Fig. 2). Finally, we need to compute the tree $T(H(h))$ for the set $H(h)$, which is comprised of all disks of H vertically below p_h since p_h is the lowest point of $P(l)$. We compute $T(H(h))$ as follows.

First, starting from an empty tree, for each $k = t, t-1, \ldots, j$ in this order, we merge $T(H(h))$ with the tree $T(H(i_k))$. Notice that the upper edge of each disk in $T(H(i_k))$ is higher than the upper edges of all disks of $T(H(h))$. Therefore, each such merge operation can be done in $O(\log m)$ time. Second, for the tree $T(H(i_{j-1}))$, we perform a split operation to split the disks into those with upper edges above p_h and those below p_h, and then merge those below p_h with $T(H(h))$ while keeping those above p_h in $T(H(i_{j-1}))$. The above split and merge operations can be done in $O(\log m)$ time. Third, we remove those disks below p_h from $H(i_0)$ and insert them to $T(H(h))$. This is done by repeatedly removing the lowest disk s from $H(i_0)$ and inserting it to $T(H(h))$ until the upper edge of s is higher than p_h. This completes our construction of the tree $T(H(h))$.

The above describes our algorithm for processing the event at p_h. One can verify that all algorithm invariants still hold. The runtime of this step is $O((1+k_1+k_2)\log m)$, where k_1 is the number of points removed from $P(l)$ (the number of merge operations is at most k_1) and k_2 is the number of disks of $H(i_0)$ got removed for constructing $T(H(h))$. As we sweep the line l from left to right, once a point is removed from $P(l)$, it will not be inserted again, and thus the total sum of k_1 in the entire algorithm is at most n. Also, once a disk is removed from $H(i_0)$, it will never be inserted again, and thus the total sum of k_2 in the entire algorithm is at most m. Hence, the overall time of the algorithm is $O((n+m)\log(n+m))$. This proves Lemma 6.

3.3 The L_2 Metric

In this section we prove Lemma 7. Recall our general position assumption that no point of P is on the boundary of a disk of S. Also recall that all points of P are above L. In the L_2 metric, the two extreme points l_k and r_k of a disk s_k are unique. For a point $p_i \in P$ and a disk $s_k \in S$, we say that p_i is *vertically above* s_k if p_i is outside s_k and $x(l_k) \leq x(p_i) \leq x(r_k)$, and p_i is *vertically below* s_k if p_i is inside s_k. We also say that s_k is vertically below p_i if p_i is vertically above s_k. Lemma 14 gives an upper bound for $|\mathcal{C}|$.

Lemma 14. *$|\mathcal{C}| \leq 2(n+m)+\kappa$.*

We next describe our algorithm for computing the set $\mathcal{C}$. For each disk s_k, we refer to the half-circle of the boundary of s_k above L as the *arc* of s_k. Note that every two arcs of S intersect at most once. Below, depending on the context, s_k may also refer to its arc. Lemma 15 computes the left and right bounding couples.

Lemma 15. *All left and right bounding couples of $\mathcal{C}$ can be computed in $O((n+m)\log(n+m)+\kappa\log m)$ time.*

To compute the middle bounding pairs of $\mathcal{C}$, the algorithm is similar in spirit to that for the L_∞ case. However, it is more involved and requires new techniques due to the nature of the L_2 metric as well as the intersections of the disks of S. We sweep a vertical line l from left to right; an event happens if l encounters a point in $P \cup \{l_k, r_k \mid 1 \leq k \leq m\}$ or an intersection of two disk arcs. Let H be the set of arcs that intersect l. During the sweeping, we maintain the following information and invariants (e.g., see Fig. 3).

1. A sequence $P(l) = \{p_{i_1}, p_{i_2}, \ldots, p_{i_t}\}$ of t points to the left of l that are sorted from left to right. $P(l)$ is maintained by a balanced binary search tree $T(P(l))$.

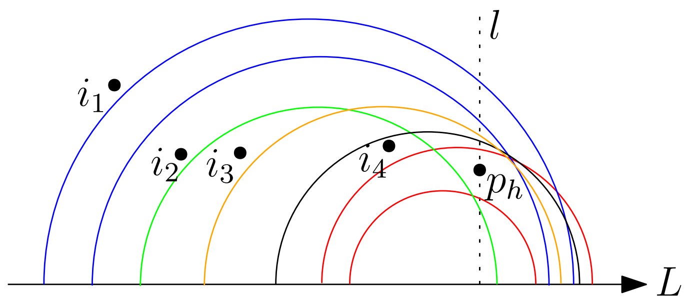

Fig. 3. In this example, $P(l) = \{p_{i_1}, p_{i_2}, p_{i_3}, p_{i_4}\}$. $H(i_1)$ consists of the two blue arcs and $H(i_4)$ consists of the two red arcs. $H(i_0)$ consists of the only black arc. After processing the event at $p_h \in P$, (i_2, h) and (i_4, h) will be reported as middle bounding couples, point i_2 will be removed from $P(l)$, and p_h will be inserted to $P(l)$. (Color figure online)

2. A collection $\mathcal{H}$ of $t+1$ subsets of H: $H(i_j)$ for $j = 0, 1, \ldots, t$, which form a partition of H, defined as follows. $H(i_t)$ is the set of disks of H vertically below p_{i_t}. For each $j = t-1, t-2, \ldots, 1$, $H(i_j)$ is the set of disks of $H \setminus \bigcup_{k=j+1}^{t} H(i_k)$ vertically below p_{i_j}. $H(i_0) = H \setminus \bigcup_{j=1}^{t} H(i_j)$. While $H(i_0)$ may be empty, none of $H(i_j)$ for $j \geq 1$ is empty.
   Each $H(i_j)$ for $0 \leq j \leq t$ is maintained by a balanced binary search tree $T(H(i_j))$ ordered by the y-coordinates of the intersections of l with the arcs of the disks. We have all disks stored in the leaves of the tree, and each internal node v of the tree stores a weight that is equal to the minimum weight of all disks in the leaves of the subtree rooted at v.
   For each subset $H' \subseteq H$, the arc of H' whose intersection with l is the lowest is called *the lowest arc* of H'. We maintain a set H^* consisting of the lowest arcs of all sets $H(i_k)$ for $1 \leq k \leq t$. So $|H^*| = t$. We use a binary search tree $T(H^*)$ to store disks of H^*, ordered by the y-coordinates of their intersections with l.
3. For each point $p_{i_j} \in P(l)$, among all points of P strictly between p_{i_j} and l, no point is vertically above any disk of $H(i_j)$.
4. Among all points of P strictly to the left of l, no point is vertically above any disk of $H(i_0)$.

Remark. Our algorithm invariants are essentially the same as those in the L_∞ case. One difference is that the points of $P(l)$ are not sorted simultaneously by y-coordinates, which is due to that the arcs of S may cross each other (in contrast, in the L_∞ case the upper edges of the squares are parallel). For the same reason, for two sets $H(i_k)$ and $H(i_j)$ with $1 \leq k < j \leq t$, it may not be the case that all arcs of $H(i_k)$ are above all arcs of $H(i_j)$ at l. Therefore, we need an additional set H^* to guide our algorithm, as will be clear later.

In our sweeping algorithm, we use similar techniques as the line segment intersection algorithm [3,4,7] to determine and handle arc intersections of S (we are able to do so because every two arcs of S intersect at most once), and the time on handling them is $O((m + \kappa) \log m)$. Below we will not explicitly explain how to handle arc intersections.

Initially $H = \emptyset$ and l is to the left of all arcs of S and all points of P.

If l encounters the left endpoint of an arc s_k, we insert s_k to $H(i_0)$.

If l encounters the right endpoint r_k of an arc s_k, then we need to determine which set of $\mathcal{H}$ contains s_k. For this, as in the L_∞ case, we associate each right endpoint with the arc. Using this mechanism, we can find the set $H(i_j)$ of $\mathcal{H}$ that contains s_k in constant time. Then, we remove s_k from $H(i_j)$. If $j = 0$, we are done for this event. Otherwise, if s_k was the lowest arc of $H(i_j)$ before the above remove operation, then s_k is also in H^* and we remove it from H^*. If the new set $H(i_j)$ becomes empty, then we remove p_{i_j} from $P(l)$. Otherwise, we find the new lowest arc from $H(i_j)$ and insert it to H^*. Processing this event takes $O(\log(n + m))$ time using the trees $T(H^*)$, $T(P(l))$, and $T(H(i_j))$.

If l encounters an intersection q of two arcs s_a and s_b, in addition to the processing work for computing the arc intersections, we do the following. Using the right endpoints, we find the two sets of $\mathcal{H}$ that contain s_a and s_b, respectively. If s_a and s_b are from the same set $H(i_j) \in \mathcal{H}$, then we switch their order in the tree $T(H(i_j))$. Otherwise, if s_a is the lowest arc in its set and s_b is also the lowest arc in its set, then both s_a and s_b are in H^*, so we switch their order in $T(H^*)$. The time for processing this event is $O(\log m)$.

If l encounters a point p_h of P, which is a major event to handle, we process it as follows. As in the L_∞ case, our goal is to determine the middle bounding couples (i, h) with $p_i \in P(l)$.

Using $T(H^*)$, we find the lowest arc s_k of H^*. Let $H(i_j)$ for some $j \in [1, t]$ be the set that contains s_k, i.e., s_k is the lowest arc of $H(i_j)$. If p_h is above s_k, then we can show that (i_j, h) is a middle bounding couple defined by and only by the arcs of $H(i_j)$ below p_h (e.g., see Fig. 3). The proof is similar to Lemma 10, so we omit the details. Hence, we report (i_j, h) as a middle bounding couple with weight equal to the minimum weight of all arcs of $H(i_j)$ below p_h, which can be found in $O(\log m)$ time using $T(H(i_j))$. Then, we split $T(H(i_j))$ into two trees by p_h such that the arcs above p_h are still in $T(H(i_j))$ and those below p_h are stored in another tree (we will discuss later how to use this tree). Next we remove s_k from H^*. If the new set $H(i_j)$ after the split operation is not empty, then we find its lowest arc and insert it into H^*; otherwise, we remove p_{i_j} from $P(l)$. We then continue the same algorithm on the next lowest arc of H^*.

The above discusses the case where p_h is above s_k. If p_h is not above s_k, we are done with processing the arcs of H^*. We can show that all middle bounding couples (b, h) with h as the right end have been computed. The proof is similar to Lemma 13, and we omit it.

Finally, we add p_h to the rear of $P(l)$. As in the L_∞ case, we need to compute the tree $T(H(h))$ for the set $H(h)$, which is comprised of all arcs of H below p_h, as follows.

Initially we have an empty tree $T(H(h))$. Let H' be the subset of the arcs of H^* vertically below p_h; here H^* refers to the original set at the beginning of the event for p_h. The set H' has already been computed above. Let $\mathcal{H}'$ be the subcollection of $\mathcal{H}$ whose lowest arcs are in H'. We process the subsets $H(i_j)$ of $\mathcal{H}'$ in the inverse order of their indices (for this, after identifying $\mathcal{H}'$, we can sort the subsets $H(i_j)$ of $\mathcal{H}'$ by their indices in $O(|H'|\log m)$ time; note that $|H'| = |\mathcal{H}'|$), i.e., the subset of $\mathcal{H}'$ with the largest index is processed first.

Suppose we are processing a subset $H(i_j)$ of $\mathcal{H}'$. Let s be the lowest arc of $H(i_j)$. Recall that we have performed a split operation on the tree $T(H(i_j))$ to obtain another tree consisting of all arcs of $H(i_j)$ below p_h, and we use $H'(i_j)$ to denote the set of those arcs and use $T(H'(i_j))$ to denote the tree. If $T(H(h))$ is empty, then we simply set $T(H(h)) = T(H'(i_j))$. Otherwise, we find the highest arc s' of $T(H(h))$ at l. If s is above s' at l, then every arc of $T(H'(i_j))$ is above all arcs of $T(H(h))$ at l and thus we simply perform a merge operation to merge $T(H'(i_j))$ with $T(H(h))$ (and we use $T(H(h))$ to refer to the new merged tree). Otherwise, we call (s, s') an *order-violation pair*. In this case, we do the following. We remove s from $T(H'(i_j))$ and insert it to $T(H(h))$. If $T(H'(i_j))$ becomes empty, then we finish processing $H(i_j)$. Otherwise, we find the new lowest arc of $T(H'(i_j))$, still denoted by s, and then process s in the same way as above.

The above describes our algorithm for processing a subset $H(i_j)$ of $\mathcal{H}'$. Once all subsets of $\mathcal{H}'$ are processed, the tree $T(H(h))$ for the set $H(h)$ is obtained.

After processing the arcs of H^* as above, we also need to consider the arcs of $H(i_0)$. For this, we scan the arcs from low to high using $T(H(i_0))$, and for each arc s, if s is above p_h, then we stop the procedure; otherwise, we remove s from $T(H(i_0))$ and insert it to $T(H(h))$.

This finishes our algorithm for processing the event at p_h. One can verify that the time complexity of this step is $O((1 + k_1 + k_2 + k_3) \cdot \log m)$ time, where k_1 is the number of middle bounding couples reported (the number of merge and split operations is at most k_1; also, $|H'| = k_1$), k_2 is the number of arcs of $H(i_0)$ got removed for constructing $T(H(h))$, and k_3 is the number of order-violation pairs. By Lemma 14, the total sum of k_1 is at most $2(n + m) + \kappa$ in the entire algorithm. As in the L_∞ case, the total sum of k_2 is at most m in the entire algorithm. The following lemma proves that the total sum of k_3 is at most κ. Therefore, the overall time of the algorithm is $O((n + m)\log(n + m) + \kappa \log m)$.

Lemma 16. *The total number of order-violation pairs in the entire algorithm is at most κ.*

References

1. Alt, H., et al.: Minimum-cost coverage of point sets by disks. In: Proceedings of the 22nd Annual Symposium on Computational Geometry (SoCG), pp. 449–458 (2006)
2. Ambühl, C., Erlebach, T., Mihalák, M., Nunkesser, M.: 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), pp. 3–14 (2006)

3. Bentley, J., Ottmann, T.: Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. **28**(9), 643–647 (1979)
4. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry – Algorithms and Applications, 3rd edn. Springer-Verlag, Berlin (2008)
5. Bilò, V., Caragiannis, I., Kaklamanis, C., Kanellopoulos, P.: Geometric clustering to minimize the sum of cluster sizes. In: Proceedings of the 13th European Symposium on Algorithms, pp. 460–471 (2005)
6. Biniaz, A., Bose, P., Carmi, P., Maheshwari, A., Munro, I., Smid, M.: Faster algorithms for some optimization problems on collinear points. In: Proceedings of the 34th International Symposium on Computational Geometry (SoCG), pp. 1–14 (2018)
7. Brown, K.: Comments on Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. **30**, 147–148 (1981)
8. Chan, T., Grant, E.: Exact algorithms and APX-hardness results for geometric packing and covering problems. Comput. Geom. Theory Appl. **47**, 112–124 (2014)
9. Claude, F., et al.: An improved line-separable algorithm for discrete unit disk cover. Discrete Math. Algorithms Appl. **2**, 77–88 (2010)
10. Claude, F., Dorrigiv, R., Durocher, S., Fraser, R., López-Ortiz, A., Salinger, A.: Practical discrete unit disk cover using an exact line-separable algorithm. In: Proceedings of the 20th International Symposium on Algorithm and Computation (ISAAC), pp. 45–54 (2009)
11. Feder, T., Greene, D.: Optimal algorithms for approximate clustering. In: Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC), pp. 434–444 (1988)
12. Feige, U.: A threshold of ln n for approximating set cover. J. ACM **45**, 634–652 (1998)
13. Har-Peled, S., Lee, M.: Weighted geometric set cover problems revisited. J. Comput. Geom. **3**, 65–85 (2012)
14. Hochbaum, D., Maass, W.: Fast approximation algorithms for a nonconvex covering problem. J. Algorithms **3**, 305–323 (1987)
15. Karmakar, A., Das, S., Nandy, S., Bhattacharya, B.: Some variations on constrained minimum enclosing circle problem. J. Comb. Optim. **25**(2), 176–190 (2013)
16. Lev-Tov, N., Peleg, D.: Polynomial time approximation schemes for base station coverage with minimum total radii. Comput. Netw. **47**, 489–501 (2005)
17. Li, J., Jin, Y.: A PTAS for the weighted unit disk cover problem. In: Proceedings of the 42nd International Colloquium on Automata, Languages and Programming (ICALP), pp. 898–909 (2015)
18. Lund, C., Yannakakis, M.: On the hardness of approximating minimization problems. J. ACM **41**, 960–981 (1994)
19. Mustafa, N., Ray, S.: PTAS for geometric hitting set problems via local search. In: Proceedings of the 25th Annual Symposium on Computational Geometry (SoCG), pp. 17–22 (2009)
20. Pedersen, L., Wang, H.: 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), pp. 158–164 (2018)
21. Wang, H., Zhang, J.: Line-constrained k-median, k-means, and k-center problems in the plane. Int. J. Comput. Geom. Appl. **26**, 185–210 (2016)