Algorithms for Covering Multiple Barriers*

Shimin Li and Haitao Wang

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

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

1 Introduction

In this paper, we study algorithms for covering multiple barriers. These are basic geometric problems and have applications in barrier coverage of mobile sensors in wireless sensor networks. For convenience, in the following we introduce and discuss the problems from the mobile sensor barrier coverage point of view.

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

We assume that covering range of the sensors is long enough so that a coverage of all barriers is always possible. Note that we can check whether a coverage is possible in $O(m+n)$ time by an easy greedy algorithm.

* This research was supported in part by NSF under Grant CCF-1317143.

F. Ellen et al. (Eds.): WADS 2017, LNCS 10389, pp. 533–544, 2017.
DOI: 10.1007/978-3-319-62127-2_45

Previously, only the special case $m = 1$ was studied and the problem was solved in $O(n^3 \log n)$ time [10]. In this paper, we propose an $O(n^2 \log n \log\log n + nm \log m)$-time algorithm for any value m, which improves the algorithm in [10] by almost a linear factor even for the case $m = 1$.

We further consider a *line-constrained* version of the problem where all sensors of S are initially on L. Previously, only the special case $m = 1$ was studied and the problem was solved in $O(n \log n)$ time [3]. We present an $O((n + m) \log(n + m))$ time algorithm for any value m, and the running time matches that of the algorithm in [3] when $m = 1$.

1.1 Related Work

Sensors are basic units in wireless sensor networks. The advantage of allowing the sensors to be mobile increases monitoring capability compared to those static ones. One of the most important applications in mobile wireless sensor networks is to monitor a barrier to detect intruders in an attempt to cross a specific region. Barrier coverage [9, 10], which guarantees that every movement crossing a barrier of sensors will be detected, is known to be an appropriate model of coverage for such applications. Mobile sensors have limited battery power and therefore their movements should be as small as possible.

Dobrev et al. [7] studies several problems on covering multiple barriers in the plane. They showed that these problems are generally NP-hard when sensors have different ranges. They also proposed polygonal-time algorithms for some special cases, e.g., barriers are parallel or perpendicular to each other, and sensors have some constrained movements. In fact, if sensors have different ranges, by an easy reduction from the Partition Problem as in [7], we can show that our problem MBC is NP-hard even for the line-constrained version and $m = 2$.

Other previous work has been focused on the line-constrained problem with $m = 1$. Czyzowicz et al. [5] first gave an $O(n^2)$ time algorithm, and later, Chen et al. [3] solved the problem in $O(n \log n)$ time. If sensors have different ranges, Chen et al. [3] presented an $O(n^2 \log n)$ time algorithm. For the *weighted case* where sensors have weights such that the moving cost of a sensor is its moving distance times its weight, Wang and Zhang [15] gave an $O(n^2 \log n \log\log n)$ time algorithm for the case where sensors have the same range.

The *min-sum* version of the line-constrained problem with $m = 1$ has also been studied, which is to minimize the sum of the moving distances of all sensors. If sensors have different ranges, the problem is NP-hard [6]. Otherwise, Czyzowicz et al. [6] gave an $O(n^2)$ time algorithm, and Andrews and Wang [1] solved the problem in $O(n \log n)$ time. The *min-num* version of the problem was also studied, where the goal is to move the minimum number of sensors to form a barrier coverage. Mehrandish et al. [13, 14] proved that the problem is NP-hard if sensors have different ranges and gave polynomial time algorithms otherwise.

Bhattacharya et al. [2] studied a circular barrier coverage problem in which the barrier is a circle and the sensors are initially located inside the circle. The goal is to move sensors to the circle to form a regular n-gon (so as to cover the

circle) such that the maximum sensor movement is minimized. An $O(n^{3.5} \log n)$-time algorithm was given in [2] and later Chen et al. [4] improved the algorithm to $O(n \log^3 n)$ time. The min-sum version of the problem was also studied [2, 4].

1.2 Our Approach

To solve MBC, one major difficulty is that we do not know the order of the sensors of S on L in an optimal solution. Therefore, our main effort is to find such an order. To this end, we first develop a *decision algorithm* that can determine whether $\lambda \geq \lambda^*$ for any value λ, where λ^* is the maximum sensor movement in an optimal solution. Our decision algorithm runs in $O(m + n \log n)$ time. Then, we solve the problem MBC by "parameterizing" the decision algorithm in a way similar in spirit to parametric search [12]. The high-level scheme of our algorithm is very similar to those in [3, 15], but many low-level computations are different.

The line-constrained problem is easier due to an *order preserving property*: there exists an optimal solution in which the order of the sensors is the same as in the input. This leads to a linear-time decision algorithm using the greedy strategy. Also based on this property, we can find a set Λ of $O(n^2 m)$ "candidate values" such that Λ contains λ^*. To avoid computing Λ explicitly, we implicitly organize the elements of Λ into $O(n)$ sorted arrays such that each array element can be found in $O(\log m)$ time. Finally, by applying the matrix search technique in [8], along with our linear-time decision algorithm, we compute λ^* in $O((n + m) \log(n+m))$ time. We should point out that implicitly organizing the elements of Λ into sorted arrays is the key and also the major difficulty for solving the problem, and our technique may be interesting in its own right.

The remaining paper is organized as follows. In Section 2, we introduce some notation. In Section 3, we present our algorithm for the line-constrained problem. In Section 4, we present our decision algorithm for MBC. Section 5 solves the problem MBC. Section 6 concludes the paper, with remarks that our techniques can be used to reduce the space complexities of the algorithms in [3, 15]. Due to the space limit, some proofs are omitted but can be found in the full paper [11].

2 Preliminaries

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

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

We denote the sensors of S by $s_1, s_2, \ldots, s_n$ sorted by their x-coordinates. For each sensor s_i located on a point x of L, $x - r$ and $x + r$ are the left and

right endpoints of the covering interval of s_i, respectively, and we call them the *left and right extensions* of s_i, respectively.

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

3 The Line-Constrained Version of MBC

In this section, we present our algorithm for the line-constrained MBC. As in the special case $m = 1$ [5], a useful observation is that the following *order preserving* property holds: There exists an optimal solution in which the order of the sensors is the same as in the input. Due to this property, we have the following lemma.

Lemma 1. *Given any $\lambda > 0$, we can determine whether λ is a feasible value in $O(n + m)$ time.*

Let OPT be an optimal solution that preserves the order of the sensors. For each $i \in [1, n]$, let x'_i be the position of s_i in OPT. We say that a set of k sensors are in *attached positions* if the union of their covering intervals is a single interval of length equal to $2rk$. The following lemma is self-evident and is an extension of a similar observation for the case $m = 1$ in [5].

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

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

By Lemma 2, λ^* is in Λ and is actually the smallest feasible value of Λ. Hence, we can first compute Λ and then find the smallest feasible value in Λ by using the decision algorithm. However, that would take $\Omega(mn^2)$ time. To reduce the time, we will not compute Λ explicitly, but implicitly organize the elements of Λ into certain sorted arrays and then apply the matrix search technique in [8]. Since we only need to deal with sorted arrays instead of more general matrices, we review the technique with respect to arrays in the following lemma.

Lemma 3. [8] *Given a set of N sorted arrays of size at most M each, we can compute the smallest feasible value of these arrays with $O(\log N + \log M)$ feasibility tests and the total time of the algorithm excluding the feasibility tests is $O(\tau \cdot N \cdot \log \frac{2M}{N})$, where τ is the time for evaluating each array element (i.e., the number of array elements that need to be evaluated is $O(N \cdot \log \frac{2M}{N})$).*

With Lemma 3, we can compute the smallest feasible values in the sets Λ_1, Λ_2, and Λ_3, respectively, and then return the smallest one as λ^*. For Λ_3, Chen et al. [3] (see Lemma 14) gave an approach to order in $O(n \log n)$ time the elements of Λ_3 into $O(n)$ sorted arrays of $O(n)$ elements each such that each array element can be obtained in $O(1)$ time. Consequently, by applying Lemma 3, the smallest feasible value of Λ_3 can be computed in $O((n+m)\log n)$ time.

For the other two sets Λ_1 and Λ_2, in the case $m = 1$, the elements of each set can be easily ordered into $O(n)$ sorted arrays of $O(n)$ elements each [3]. However, in our problem for general m, it becomes significantly more difficult to obtain a subquadratic-time algorithm. Indeed, this is the main challenge of our method. In what follows, our main effort is to prove Lemma 4.

Lemma 4. *For the set Λ_1, in $O(m \log m)$ time, we can implicitly form a set $\mathcal{A}$ of $O(n)$ sorted arrays of $O(m^2 n)$ elements each such that each array element can be computed in $O(\log m)$ time and every element of Λ_1 is contained in one of the arrays. The same applies to the set Λ_2.*

We note that our technique for Lemma 4 might be interesting in its own right and may find other applications as well. Before proving Lemma 4, we first prove the following theorem by using Lemma 4.

Theorem 1. *The line-constrained version of MBC can be solved in $O((n+m)\log(n+m))$ time.*

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

Let Λ_1' denote the set of all elements in the arrays of $\mathcal{A}$ specified in Lemma 4. Define Λ_2' similarly with respect to Λ_2. By Lemma 4, $\Lambda_1 \subseteq \Lambda_1'$ and $\Lambda_2 \subseteq \Lambda_2'$. Since $\lambda^* \in \Lambda_1 \cup \Lambda_2 \cup \Lambda_3$, we also have $\lambda^* \in \Lambda_1' \cup \Lambda_2' \cup \Lambda_3$. Hence, λ^* is the smallest feasible value in $\Lambda_1' \cup \Lambda_2' \cup \Lambda_3$. Let λ_1, λ_2, and λ_3 be the smallest feasible values in the sets Λ_1', Λ_2', and Λ_3, respectively. As discussed before, λ_3 can be computed in $O((n+m)\log n)$ time. By Lemma 4, applying the algorithm in Lemma 3 can compute both λ_1 and λ_2 in $O((n+m)(\log m + \log n))$ time. Note that $(n+m)(\log m + \log n) = \Theta((n+m)\log(n+m))$. □

3.1 Proving Lemma 4

In this section, we prove Lemma 4. We will only prove the case for the set Λ_1, since the other case for Λ_2 is symmetric. Recall that $\Lambda_1 = \{\lambda(i,j,k) \mid 1 \le i \le j \le n, 1 \le k \le m\}$, where $\lambda(i,j,k) = x_j - (a_k + 2r(j-i) + r)$.

For any j and k, let $A[j,k]$ denote the list $\lambda(i,j,k)$ for $i = 1, 2, \ldots, j$, which is sorted increasingly. Let $A[j]$ denote the union of the elements in $A[j,k]$ for all $k \in [1,m]$. Clearly, $\Lambda_1 = \bigcup_{j=1}^{n} A[j]$. In the following, we will organize the elements in each $A[j]$ into a sorted array $B[j]$ of size $O(nm^2)$ such that given any index t, the t-th element of $B[j]$ can be computed in $O(\log m)$ time, which will prove Lemma 4. Our technique replies on the following property: the difference of every two adjacent elements in each list $A[j,k]$ is the same, i.e., $2r$.

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

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

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

Lemma 5. *For any t with $1 \le t \le \alpha[j] \cdot m$, the t-th smallest element of $B[j]$ is the h_t-th element of the list $B[j, \sigma_j(k_t)]$, where $h_t = \lceil \frac{t}{m} \rceil$ and $k_t = t \mod m$.*

By Lemma 5, if σ_j is known, we can obtain the t-th smallest element of $B[j]$ in $O(1)$ time for any t. Computing σ_j can be done in $O(m \log m)$ time by sorting. If we do the sorting for every $j \in [1,n]$, then we wound need $O(nm \log m)$ time. Fortunately, Lemma 6 implies that we only need to do the sorting once.

Lemma 6. *The permutation σ_j is unique for all $j \in [1,n]$.*

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

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

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

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

Consider any group G_g with $1 \leq g \leq l$. We process the lists $A[j][k]$ for all $k \in G_g$ in the same way as discussed before. Specifically, for each $k \in G_g$, we create a new list $B[j][k]$ from $A[j][k]$. Based on the new lists in the group G_g, we form the sorted array $B_g[j]$ with a total of $|G_g| \cdot \alpha_g[j]$ elements, where $|G_g|$ is the number of indices of G_g and $\alpha_g[j]$ is corresponding $\alpha[j]$ value as defined before but only on the group G_g, i.e., if k_1 and k_2 are the smallest and largest indices of G_g respectively, then $\alpha_g[j] = 1 + \lceil \frac{\lambda(j,j,k_1) - \lambda(1,j,k_2)}{2r} \rceil$. Let $B[j]$ be the sorted list of all elements in the lists $B_g[j]$ for all groups. Due to the intra-group overlapping property of each group, it holds that $\alpha_g \leq |G_g| \cdot n$. Thus, the size of $B[j]$ is at most $\sum_{g=1}^{l} |G_g|^2 \cdot n$, which is at most $m^2 n$ since $\sum_{g=1}^{l} |G_g| = m$.

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

The above discussion is on any single index $j \in [1, n]$. With $O(m \log m)$ time preprocessing, given any t, we can find the t-th smallest value of $B[j]$ in $O(\log m)$ time. For all indices $j \in [1, n]$, it appears that we have to do the group partition for every $j \in [1, n]$, which would take quadratic time. To resolve the issue, we show that it suffices to only use the group partition based on $j = n$ for all other $j \in [1, n-1]$. The details are given below.

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

Lemma 7. *The inter-group non-overlapping property holds for any $j \in [1, n-1]$.*

Consider any G_g with $1 \leq g \leq l$ and any $j \in [1, n]$. For each $k \in G_g$, we create a new list $B[j][k]$ based on $A[j][k]$ in the same way as before. Based on the new lists, we form the sorted array $B_g[j]$ of $|G_g| \cdot \alpha_g[j]$ elements. We also define the value $\beta_g[j]$ in the same way as before. Lemma 8 shows that $\alpha_g[j]$ and $\beta_g[j]$ can be computed from $\alpha_g[n]$ and $\beta_g[n]$.

Lemma 8. *For any $j \in [1, n-1]$ and $g \in [1, l]$, $\alpha_g[j] = \alpha_g[n] - n + j$ and $\beta_g[j] = \beta_g[n] + \delta_g \cdot g \cdot (j - n)$, where $\delta_g = \sum_{g'=1}^{g} |G_{g'}|$.*

For each group G_g, we compute the permutation for the lists $B[n, k]$ for all k in the group. Computing the permutations for all groups takes $O(m \log m)$ time. Also as preprocessing, we first compute δ_g, $\alpha_g(n)$ and $\beta_g(n)$ for all $g \in [1, l]$ in $O(m)$ time. By Lemma 8, for any $j \in [1, n]$ and any $g \in [1, l]$, we can compute $\alpha_g[j]$ and $\beta_g[j]$ in $O(1)$ time. Because the lists $B[n, k]$ for all k in each group G_g have the intra-group overlapping property, it holds that $\alpha_g[n] \leq |G_g| \cdot n$. Hence, $\sum_{g=1}^{l} \alpha_g[n] \leq mn$. For any $j \in [1, n-1]$, by Lemma 8, $\alpha_g[j] < \alpha_g[n]$, and thus $\sum_{g=1}^{l} \alpha_g[j] \leq mn$. Note that $B[j]$ has at most $m^2 n$ elements.

For any $j \in [1, n]$ and any $t \in [1, \sum_{g=1}^{l} |G_g| \cdot \alpha_g[j]]$, to compute the t-th smallest element of $B[j]$, due to the inter-group non-overlapping property in Lemma 7, we can still use the previous binary search approach. As we can obtain each $\beta_g[j]$ for any $g \in [1, l]$ in $O(1)$ time by Lemma 8, we can still compute the t-th smallest element of $B[j]$ in $O(\log m)$ time. This proves Lemma 4.

4 The Decision Problem of MBC

In this section, we present an $O(m + n \log n)$-time algorithm for the decision problem of MBC: given any value $\lambda > 0$, determine whether $\lambda \geq \lambda^*$. Our algorithm for MBC in Section 5 will make use of this decision algorithm. The decision problem may have independent interest because in some applications each sensor has a limited energy λ and we want to know whether their energy is enough for them to move to cover all barriers.

Consider any value $\lambda > 0$. We assume $\lambda \geq \max_{1 \leq i \leq n} |y_i|$ since otherwise some sensor cannot reach L by moving λ (and thus λ is not feasible). For any sensor $s_i \in S$, define $x_i^r = x_i + \sqrt{\lambda^2 - y_i^2}$ and $x_i^l = x_i - \sqrt{\lambda^2 - y_i^2}$. Note that x_i^r and x_i^l are respectively the rightmost and leftmost points of L s_i can reach with respect to λ. We call x_i^r the *rightmost (resp., leftmost) λ-reachable location* of s_i on L. For any point x on L, we use $p^+(x)$ to denote a point x' such that $x' > x$ and x' is infinitesimally close to x. The high-level scheme of our algorithm is similar to that in [15]. Below we describe the algorithm.

We use a *configuration* to refer to a specification on where each sensor $s_i \in S$ is located. For example, in the *input configuration*, each s_i is at (x_i, y_i). We first move each sensor s_i to x_i^r on L. Let C_0 denote the resulting configuration. In C_0, each sensor s_i is not allowed to move rightwards but can move leftwards on L by a maximum distance $2\sqrt{\lambda^2 - y_i^2}$.

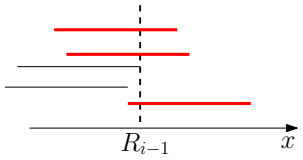

Fig. 1. Illustrating the set S_{i1}. The covering intervals of sensors are shown with segments (the red thick segments correspond to the sensors in S_{i1}). Every sensor in S_{i1} can be $s_{g(i)}$.

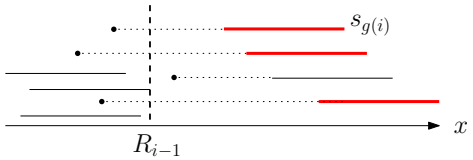

Fig. 2. Illustrating the set S_{i2}. The segments are the covering intervals of sensors. The red thick segments correspond to the sensors in S_{i2}. The four black points corresponding to the values $x_k^l - r$ of the four sensors x_k to the right of R_{i-1}. The sensor $s_{g(i)}$ is labeled.

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

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

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

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

If $S_{i1} = \emptyset$, then we define $S_{i2} = \{s_k \mid x_k^l - r \leq R_{i-1} < x_k^r - r\}$ (i.e., S_{i2} consists of those sensors s_k that does not cover R_{i-1} when it is at x_k^r but is possible to do so when it is at some location in $[x_k^l, x_k^r]$). If $S_{i2} \neq \emptyset$, we choose the *leftmost* sensor of S_{i2} as $s_{g(i)}$ (e.g., see Fig. 2), and let $x'_{g(i)} = R_{i-1} + r$ (i.e., we move $s_{g(i)}$ to $x'_{g(i)}$ and thus obtain C_i). If $S_{i2} = \emptyset$, then we conclude that $\lambda < \lambda^*$ and terminate the algorithm.

Hence, if $S_{i1} = S_{i2} = \emptyset$, the algorithm will stop and report $\lambda < \lambda^*$. Otherwise, a sensor $s_{g(i)}$ is found from either S_{i1} or S_{i2}, and it is moved to $x'_{g(i)}$. In either case, $R_i = x'_{g(i)} + r$ and $S_i = S_{i-1} \cup \{s_{g(i)}\}$. If $R_i \geq \beta$, then we terminate the algorithm and report $\lambda \geq \lambda^*$. Otherwise, we further perform the following *jump-*

over procedure: We check whether R_i is located at the interior of any barrier; if not, then we set R_i to the left endpoint of the barrier right after R_i.

This finishes the i-th step of our algorithm. One can verify that all algorithm invariants are maintained. As S has n sensors, the algorithm will finish in at most n steps. This finishes the description of our algorithm. The algorithm correctness and implementation are omitted.

Theorem 2. *Given any value λ, we can determine whether $\lambda \geq \lambda^*$ in $O(m + n \log n)$ time.*

Our algorithm in Section 5 will perform feasibility tests multiple times, for which we have the following lemma.

Lemma 9. *Suppose the values x_i^r for all $i = 1, 2, \ldots, n$ are already sorted, we can determine whether $\lambda \geq \lambda^*$ in $O(m + n \log \log n)$ time for any λ.*

5 Solving the Problem MBC

To solve MBC, it suffices to compute λ^*. The high-level scheme of our algorithm is similar to that in [15], although some low-level computations are different.

We now use $x_i^r(\lambda)$ to refer to x_i^r for any λ, so that we consider $x_i^r(\lambda)$ as a function on $\lambda \in [0, \infty]$, which actually defines a half of the upper branch (on the right side of the y-axis) of a hyperbola. Let σ be the order of the values $x_i^r(\lambda^*)$ for all $i \in [1, n]$. To use Lemma 9, we first run a preprocessing step in Lemma 10.

Lemma 10. *With $O(n \log^3 n + m \log^2 n)$ time preprocessing, we can compute σ and an interval $(\lambda_1^*, \lambda_2^*]$ containing λ^* such that σ is also the order of the values $x_i^r(\lambda)$ for any $\lambda \in (\lambda_1^*, \lambda_2^*]$.*

Proof. To compute σ, we apply Megiddo's parametric search [12] to sort the values $x_i^r(\lambda^*)$ for $i \in [1, n]$, using the decision algorithm in Theorem 2. Indeed, recall that $x_i^r(\lambda) = x_i + \sqrt{\lambda^2 - y_i^2}$. Hence, as λ increases, $x_i^r(\lambda)$ is a (strictly) increasing function. For any two indices i and j, there is at most one root on $\lambda \in [0, \infty)$ for the equation: $x_i^r(\lambda) = x_j^r(\lambda)$. Therefore, we can apply Megiddo's parametric search [12] to do the sorting. The total time is $O((\tau + n) \log^2 n)$, where τ is the running time of the decision algorithm. By Theorem 2, $\tau = O(m + n \log n)$. Hence, the total time for computing σ is $O(m \log^2 n + n \log^3 n)$.

In addition, Megiddo's parametric search [12] will return an interval $(\lambda_1^*, \lambda_2^*]$ containing λ^* and σ is also the order of the values $x_i^r(\lambda)$ for any $\lambda \in (\lambda_1^*, \lambda_2^*]$. □

As $\lambda^* \in (\lambda_1^*, \lambda_2^*]$, our subsequent feasible tests will be only on values $\lambda \in (\lambda_1^*, \lambda_2^*)$ because if $\lambda \leq \lambda_1^*$, then λ is not feasible and if $\lambda \geq \lambda_2^*$, then λ is feasible. Lemmas 9 and 10 together lead to the following result.

Lemma 11. *Each feasibility test can be done in $O(m + n \log \log n)$ time for any $\lambda \in (\lambda_1^*, \lambda_2^*)$.*

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

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

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

Initially when $i = 1$, we let $\lambda_0^1 = \lambda_1^*$ and $\lambda_0^2 = \lambda_2^*$. Since $S_0(\lambda) = \emptyset$ and $R_0(\lambda) = 0$ for any λ, by Lemma 10, all invariants hold for $i = 1$. In general, the i-th step will either compute λ^*, or obtain an interval $(\lambda_i^1, \lambda_i^2] \subseteq (\lambda_{i-1}^1, \lambda_{i-1}^2]$ and a sensor $s_{g(i)}(\lambda)$ with $S_i(\lambda) = S_{i-1}(\lambda) \cup \{s_{g(i)}(\lambda)\}$. The running time of the step is $O((m + n \log\log n)(\log n + \log m))$. The details are omitted.

The algorithm will compute λ^* after at most n steps. The total time is $O(n \cdot (m + n \log\log n) \cdot (\log m + \log n))$, which is bounded by $O(nm \log m + n^2 \log n \log\log n)$ as shown in Theorem 3. The space of the algorithm is $O(n)$.

Theorem 3. *The problem MBC can be solved in $O(nm \log m + n^2 \log n \log\log n)$ time and $O(n)$ space.*

6 Concluding Remarks

As mentioned before, the high-level scheme of our algorithm for MBC is similar to those in [3, 15]. However, a new technique we propose in this paper can help reduce the space complexities of the algorithms in [3, 15]. Specifically, Chen et al. [3] solved the line-constrained problem in $O(n^2 \log n)$ time and $O(n^2)$ space for the case where $m = 1$ and sensors have different ranges. Wang and Zhang [15] solved the line-constrained problem in $O(n^2 \log n \log\log n)$ time and $O(n^2)$ space for the case where $m = 1$, sensors have the same range, and sensors have weights. If we apply the similar preprocessing as in Lemma 10, then the space complexities of both algorithms [3, 15] can be reduced to $O(n)$ while the time complexities do not change asymptotically.

In addition, by slightly changing our algorithm for MBC, we can also solve the following problem variant: Find a subset S' of sensors of S to move them to

L to cover all barriers such that the maximum movement of all sensors of S' is minimized (and sensors of $S \setminus S'$ do not move). We omit the details.

References

1. Andrews, A., Wang, H.: Minimizing the aggregate movements for interval coverage. Algorithmica 78, 47–85 (2017)
2. Bhattacharya, B., Burmester, B., Hu, Y., Kranakis, E., Shi, Q., Wiese, A.: Optimal movement of mobile sensors for barrier coverage of a planar region. Theoretical Computer Science 410(52), 5515–5528 (2009)
3. Chen, D., Gu, Y., Li, J., Wang, H.: Algorithms on minimizing the maximum sensor movement for barrier coverage of a linear domain. Discrete and Computational Geometry 50, 374–408 (2013)
4. Chen, D., Tan, X., Wang, H., Wu, G.: Optimal point movement for covering circular regions. Algorithmica 72, 379–399 (2013)
5. Czyzowicz, J., Kranakis, E., Krizanc, D., Lambadaris, I., Narayanan, L., Opatrny, J., Stacho, L., Urrutia, J., Yazdani, M.: On minimizing the maximum sensor movement for barrier coverage of a line segment. In: Proc. of the 8th International Conference on Ad-Hoc, Mobile and Wireless Networks. pp. 194–212 (2009)
6. Czyzowicz, J., Kranakis, E., Krizanc, D., Lambadaris, I., Narayanan, L., Opatrny, J., Stacho, L., Urrutia, J., Yazdani, M.: On minimizing the sum of sensor movements for barrier coverage of a line segment. In: Proc. of the 9th International Conference on Ad-Hoc, Mobile and Wireless Networks. pp. 29–42 (2010)
7. Dobrev, S., Durocher, S., Eftekhari, M., Georgiou, K., Kranakis, E., Krizanc, D., Narayanan, L., Opatrny, J., Shende, S., Urrutia, J.: Complexity of barrier coverage with relocatable sensors in the plane. Theoretical Computer Science 579, 64–73 (2015)
8. Frederickson, G., Johnson, D.: Generalized selection and ranking: Sorted matrices. SIAM Journal on Computing 13(1), 14–30 (1984)
9. Kumar, S., Lai, T., Arora, A.: Barrier coverage with wireless sensors. In: Proc. of the 11th Annual International Conference on Mobile Computing and Networking (MobiCom). pp. 284–298 (2005)
10. Li, S., Shen, H.: Minimizing the maximum sensor movement for barrier coverage in the plane. In: Proc. of the 2015 IEEE Conference on Computer Communications (INFOCOM). pp. 244–252 (2015)
11. Li, S., Wang, H.: Algorithms for covering multiple barriers. arXiv:1704.06870 (2017)
12. Megiddo, N.: Applying parallel computation algorithms in the design of serial algorithms. Journal of the ACM 30(4), 852–865 (1983)
13. Mehrandish, M.: On Routing, Backbone Formation and Barrier Coverage in Wireless Ad Doc and Sensor Networks. Ph.D. thesis, Concordia University, Montreal, Quebec, Canada (2011)
14. Mehrandish, M., Narayanan, L., Opatrny, J.: Minimizing the number of sensors moved on line barriers. In: Proc. of IEEE Wireless Communications and Networking Conference (WCNC). pp. 653–658 (2011)
15. Wang, H., Zhang, X.: Minimizing the maximum moving cost of interval coverage. In: Proc. of the 26th International Symposium on Algorithms and Computation (ISAAC). pp. 188–198 (2015), full version to appear in *International Journal of Computational Geometry and Application (IJCGA)*