Near-linear Time Approximation Schemes for Geometric Maximum Coverage[☆]

Kai Jin[a,1], Jian Li[b,*], Haitao Wang[c,2], Bowei Zhang[b,3], Ningye Zhang[b,4]

[a] *Department of Computer Science, University of Hong Kong, Hong Kong SAR*
[b] *Institute for Interdisciplinary Information Sciences (IIIS), Tsinghua University, Beijing, China, 100084*
[c] *Department of Computer Science, Utah State University, Utah, USA, 84322*

Abstract

We study approximation algorithms for the following geometric version of the maximum coverage problem: Let $\mathcal{P}$ be a set of n weighted points in the plane. Let D represent a planar object, such as a rectangle, or a disk. We want to place m copies of D such that the sum of the weights of the points in $\mathcal{P}$ covered by these copies is maximized. For any fixed $\varepsilon > 0$, we present efficient approximation schemes that can find a $(1-\varepsilon)$-approximation to the optimal solution. In particular, for $m = 1$ and for the special case where D is a rectangle, our algorithm runs in time $O(n\log(\frac{1}{\varepsilon}))$, improving on the previous result. For $m > 1$ and the rectangular case, our algorithm runs in $O(\frac{n}{\varepsilon}\log(\frac{1}{\varepsilon}) + \frac{m}{\varepsilon}\log m + m(\frac{1}{\varepsilon})^{O(\min(\sqrt{m},\frac{1}{\varepsilon}))})$ time. For a more general class of shapes (including disks, polygons with $O(1)$ edges), our algorithm runs in $O(n(\frac{1}{\varepsilon})^{O(1)} + \frac{m}{\epsilon}\log m + m(\frac{1}{\varepsilon})^{O(\min(m,\frac{1}{\varepsilon^2}))})$ time.

Keywords: Maximum Coverage, Geometric Set Cover, Polynomial-Time

---

[☆]The conference version of this paper was published in COCOON 2015: The 21st Annual International Computing and Combinatorics Conference.

[*]corresponding author: `lijian83@mail.tsinghua.edu.cn`

[1] `cskaijin@hku.hk`

[2] `haitao.wang@usu.edu`

[3] `bw-zh14@mails.tsinghua.edu.cn`

[4] `zhangny12@mails.tsinghua.edu.cn`

[5]J. Li, B. Zhang and N. Zhang's research was supported in part by the National Basic Research Program of China Grant 2015CB358700, 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61202009, 61033001, 61361136003.

[6]H. Wang's research was supported in part by NSF under Grant CCF-1317143.

1. Introduction

The maximum coverage problem is a classic problem in theoretical computer science and combinatorial optimization. In this problem, we are given a universe $\mathcal{P}$ of weighted elements, a family of subsets and a number m. The goal is to select at most m of these subsets such that the sum of the weights of the covered elements in $\mathcal{P}$ is maximized. It is well-known that the most natural greedy algorithm achieves an approximation factor of $1-1/e$, which is essentially optimal (unless P=NP) [1, 2, 3]. However, for several geometric versions of the maximum coverage problem, better approximation ratios can be achieved (we will mention some of such results below). In this paper, we mainly consider the following geometric maximum coverage problem:

Definition. ($\mathsf{MaxCov}_{\mathsf{R}}(\mathcal{P}, m)$) Let $\mathcal{P}$ be a set of n points in a 2-dimensional Euclidean plane $\mathbb{R}^2$. Each point $p \in \mathcal{P}$ has a given weight $w_p \geq 0$. The goal of our geometric max-coverage problem (denoted as $\mathsf{MaxCov}_{\mathsf{R}}(\mathcal{P}, m)$) is to place m $a \times b$ rectangles such that the sum of the weights of the covered points by these rectangles is maximized. More precisely, let S be the union of m rectangles we placed. Our goal is to maximize

$$\mathsf{Cover}(\mathcal{P}, S) = \sum_{p \in \mathcal{P} \cap S} w_p.$$

We also study the same coverage problem with other shapes, instead of rectangles. We denote the corresponding problem for circular disk as $\mathsf{MaxCov}_{\mathsf{C}}(\mathcal{P}, m)$, and denote the corresponding problem for a general object D as $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}, m)$. One natural application of the geometric maximum coverage problem is the facility placement problem. In this problem, we would like to locate a certain number of facilities to serve the maximum number of clients. Each facility can serve a region (depending on whether the metric is L_1 or L_2, the region is either a square or a disk).

1.1. $m = 1$

Previous Results: We first consider $\mathsf{MaxCov}_{\mathsf{R}}(\mathcal{P}, 1)$, i.e., the maximum coverage problem with 1 rectangle. Imai and Asano [4], Nandy and Bhattacharya [5] gave two different exact algorithms for computing $\mathsf{MaxCov}_{\mathsf{R}}(\mathcal{P}, 1)$, both running in time $O(n \log n)$. It is also known that solving $\mathsf{MaxCov_R}(\mathcal{P}, 1)$ exactly in algebraic decision tree model requires $\Omega(n \log n)$ time [6]. Tao et al. [7] proposed a randomized approximation scheme for $\mathsf{MaxCov_R}(\mathcal{P}, 1)$. With probability $1 - 1/n$, their algorithm returns a $(1 - \varepsilon)$-approximate answer in $O(n \log(\frac{1}{\varepsilon}) + n \log \log n)$ time. In the same paper, they also studied the problem in the external memory model.

Our Results: For $\mathsf{MaxCov_R}(\mathcal{P}, 1)$ we show that there is an approximation scheme that produces a $(1 - \varepsilon)$-approximation and runs in $O(n \log(\frac{1}{\varepsilon}))$ time, improving the result by Tao et al. [7].

1.2. General $m > 1$

Previous Results: Both $\mathsf{MaxCov_R}(\mathcal{P}, m)$ and $\mathsf{MaxCov_C}(\mathcal{P}, m)$ are NP-hard if m is part of the input [8]. The most related work is de Berg, Cabello and Har-Peled [9]. They mainly focused on using unit disks (i.e., $\mathsf{MaxCov_C}(\mathcal{P}, m)$). They proposed a $(1 - \varepsilon)$-approximation algorithm for $\mathsf{MaxCov_C}(\mathcal{P}, m)$ with time complexity $O(n(m/\varepsilon)^{O(\sqrt{m})})$. [7] We note that their algorithm can be easily extended to $\mathsf{MaxCov_R}$ with the same time complexity.

We are not aware of any explicit result for $\mathsf{MaxCov_R}(\mathcal{P}, m)$ for general $m > 1$. It is known [9] that the problem admits a PTAS via the standard shifting technique [10]. [8]

Our Results: Our main result is an approximation scheme for $\mathsf{MaxCov_R}(\mathcal{P}, m)$ which runs in time

$$O\left(\frac{n}{\varepsilon} \log \frac{1}{\varepsilon} + \frac{m}{\varepsilon} \log m + m \left(\frac{1}{\varepsilon}\right)^{\Delta_1}\right),$$

where $\Delta_1 = O(\min(\sqrt{m}, \frac{1}{\varepsilon}))$. Our algorithm can also be extended to other shapes subject to some common assumptions, including disks, polygons with $O(1)$ edges (see Section 5 for the assumptions). The running time of our

[7] They were mainly interested in the case where m is a constant. So the running time becomes $O(n(1/\varepsilon)^{O(\sqrt{m})})$ (which is the bound claimed in their paper) and the exponential dependency on m does not look too bad for $m = O(1)$. Since we consider the more general case, we make the dependency on m explicit.

[8] Hochbaum and Maass [10] obtained a PTAS for the problem of covering given points with a minimal number of rectangles. Their algorithm can be easily modified into a PTAS for $\mathsf{MaxCov_R}(\mathcal{P}, m)$ with running time $n^{O(1/\epsilon)}$.

algorithm is

$$O\left(n\Big(\frac{1}{\varepsilon}\Big)^{O(1)} + \frac{m}{\varepsilon}\log m + m\Big(\frac{1}{\varepsilon}\Big)^{\Delta_2}\right),$$

where $\Delta_2 = O(\min(m, \frac{1}{\varepsilon^2}))$.

Following the convention of approximation algorithms, ε is a fixed constant. Hence, the second and last term is essentially $O(m \log m)$ and the overall running time is essentially linear $O(n)$ (if $m = O(n/\log n)$).

Our algorithm follows the standard shifting technique [10], which reduces the problem to a smaller problem restricted in a constant size cell. The same technique is also used in de Berg et al. [9]. They proceeded by first solving the problem exactly in each cell, and then use dynamic programming to find the optimal allocation for all cells. [9]

Our improvement comes from another two simple yet useful ideas. First, we apply the shifting technique in a different way and make the side length of grids much smaller ($O(\frac{1}{\varepsilon})$, instead of $O(m)$ in de Berg et al.'s algorithm [9]). Second, we solve the dynamic program approximately. In fact, we show that a simple greedy strategy (along with some additional observations) can be used for this purpose, which allows us to save another $O(m)$ term.

1.3. Other Related Work

There are many different variants for this problem. We mention some most related problems here.

Barequet et al. [11], Dickerson and Scharstein [12] studied the max-enclosing polygon problem which aims to find a position of a given polygon to cover maximum number of points. This is the same as $\mathsf{MaxCov}_\mathsf{R}(\mathcal{P}, 1)$ if the polygon is a rectangle. Imai et al. [4] gave an optimal algorithm for the max-enclosing rectangle problem with time complexity $O(n \log n)$.

$\mathsf{MaxCov}_\mathsf{C}(\mathcal{P}, m)$ was introduced by Drezner [13]. Chazelle and Lee [14] gave an $O(n^2)$-time exact algorithm for the problem $\mathsf{MaxCov}_\mathsf{C}(\mathcal{P}, 1)$. A Monte-Carlo $(1-\varepsilon)$-approximation algorithm for $\mathsf{MaxCov}_\mathsf{C}(\mathcal{P}, 1)$ was shown in [15], where $\mathcal{P}$ is an unweighted point set. Aronov and Har-Peled [16] showed that for unweighted point sets an $O(n\varepsilon^{-2}\log n)$ time Monte-Carlo $(1-\varepsilon)$-approximation algorithm exists, and also provided some results for

[9] In fact, their dynamic programming runs in time at least $\Omega(m^2)$. Since they focused on constant m, this term is negligible in their running time. But if $m > \sqrt{n}$, the term can not be ignored and may become the dominating term.

other shapes. de Berg et al. [9] provided an $O(n\varepsilon^{-3})$ time $(1-\varepsilon)$-approximation algorithm.

For $m > 1$, $\mathsf{MaxCov_C}(\mathcal{P}, m)$ has only a few results. For $m = 2$, Cabello et al. [17] gave an exact algorithm for this problem when the two disks are disjoint in $O(n^{8/3}\log^2 n)$ time. de Berg et al. [9] gave $(1-\varepsilon)$-approximation algorithms that run in $O(n\varepsilon^{-4m+4}\log^{2m-1}(1/\varepsilon))$ time for $m > 3$ and in $O(n\varepsilon^{-6m+6}\log(1/\varepsilon))$ time for $m = 2, 3$.

The dual of the maximum coverage problem is the classical set cover problem. The geometric set cover problem has enjoyed extensive study in the past two decades. The literature is too vast to list exhaustively here. See e.g., [18, 19, 20, 21, 22, 23, 24] and the references therein.

Outline. We consider the rectangular case first, and then show the extension to general shapes in the last section.

2. Preliminaries

We first define some notations and mention some results that are needed in our algorithm. Denote by $G_\delta(a,b)$ the square grid with mesh size δ such that the vertical and horizontal lines are defined as follows

$$\begin{aligned} G_\delta(a,b) = \quad & \{(x,y) \in \mathbb{R}^2 \mid y = b + k\cdot\delta, k \in \mathbb{Z}\} \\ & \cup \{(x,y) \in \mathbb{R}^2 \mid x = a + k\cdot\delta, k \in \mathbb{Z}\}\,. \end{aligned}$$

Given $G_\delta(a,b)$ and a point $p = (x,y)$, we call the integer pair $(\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor)$ the *index* of p (the index of the cell in which p lies in).

Perfect Hashing: Dietzfelbinger et al. [25] shows that if each basic algebraic operation (including $\{+,-,\times,\div,\log_2,\exp_2\}$) can be done in constant time, we can get a perfect hash family so that each insertion and membership query takes $O(1)$ expected time. In particular, using this hashing scheme, we can hash the indices of all points, so that we can obtain the list of all non-empty cells in $O(n)$ expected time. Moreover, for any non-empty cell, we can retrieve all points lies in it in time linear in the number of such points.

Linear Time Weighted Median and Selection: It is well known that finding the weighted median for an array of numbers can be done in deterministic worst-case linear time. The setting is as follows: Given n distinct elements $x_1, x_2, ..., x_n$ with positive weights $w_1, w_2, ..., w_n$. Let $w = \sum_{i=1}^{n} w_i$. The *weighted median* is the element x_k satisfying $\sum_{x_i<x_k} w_i < w/2$ and

$\sum_{x_i > x_k} w_i \leq w/2$. Finding the k-th smallest elements for any array can also be done in deterministic worst-case linear time. See e.g., [26].

An Exact Algorithm for $\mathsf{MaxCov_R}(\mathcal{P}, 1)$: As we mentioned, Nandy and Bhattacharya [5] provided an $O(n \log n)$ exact algorithm for the $\mathsf{MaxCov_R}(\mathcal{P}, 1)$ problem. We use this algorithm as a subroutine in our algorithm.

3. A Linear Time Algorithm for $\mathsf{MaxCov_R}(\mathcal{P}, 1)$

Notations: Without loss of generality, we can assume that $a = b = 1$, i.e., all the rectangles are 1×1 squares, (by properly scaling the input). We also assume that all points are in general positions. In particular, all coordinates of all points are distinct. For a unit square r, we use $w(r)$ to denote the sum of the weights of the points covered by r. We say a unit square r is located at (x, y) if the top-left corner of r is (x, y). We regard r as a closed set; so it contains its boundaries.

Now we present our approximation algorithm for $\mathsf{MaxCov_R}(\mathcal{P}, 1)$.

3.1. Grid Shifting

Recall the definition of a grid $G_\delta(a, b)$ (in Section 2). Consider the following four grids: $G_2(0, 0)$, $G_2(0, 1)$, $G_2(1, 0)$, $G_2(1, 1)$ with $\delta = 2$. We can easily see that for any unit square r, there exists one of the above grids that does not intersect r (i.e., r is inside some cell of the grid). This is also the case for the optimal solution.

Now, we describe the overall framework, which is similar to that in [7]. Our algorithm differs in several details. MaxCovCell(c) is a subroutine that takes a 2×2 cell c as input and returns a unit square r that is a (1-ε)-approximate solution if the problem is restricted to cell c. We present the details of MaxCovCell in the next subsection.

As we argued above, there exists a grid G such that the optimal solution is inside some cell $\mathsf{c}^\star \in G$. Therefore, MaxCovCell($\mathsf{c}^\star$) should return a (1-ε)-approximation for the original problem $\mathsf{MaxCov_R}(\mathcal{P}, 1)$.

3.2. MaxCovCell

In this section, we present the details of the subroutine MaxCovCell. Now we are dealing with the problem restricted to a single 2×2 cell c. Denote the number of point in c by n_c, and the sum of the weights of points in c by W_c. We distinguish two cases, depending on whether n_c is larger or

Algorithm 1 $\mathsf{MaxCov_R}(\mathcal{P}, 1)$

$w_{\max} \leftarrow 0$
for each $G \in \{G_2(0,0), G_2(0,1), G_2(1,0), G_2(1,1)\}$ **do**
  Use perfect hashing to find all the non-empty cells of G.
  for each non-empty cell c of G **do**
    $r \leftarrow$ MaxCovCell(c).
    If $w(r) > w_{\max}$, **then** $w_{\max} \leftarrow w(r)$ and $r_{\max} \leftarrow r$.
  end for;
end for;
return $r_{\max}$;

smaller than $\left(\frac{1}{\varepsilon}\right)^2$. If $n_{\mathsf{c}} < \left(\frac{1}{\varepsilon}\right)^2$, we simply apply the $O(n \log n)$ time exact algorithm. [5]

The other case requires more work. In this case, we further partition cell c into many smaller cells. First, we need the following simple lemma.

Lemma 1. *Given n points in $\mathbb{R}^2$ with positive weights $w_1, w_2, ..., w_n$, $\sum_{i=1}^{n} w_i = w$. Assume that $x_1, x_2, ..., x_n$ are their distinct x-coordinates. We are also given a value w_d such that $\max(w_1, w_2, ..., w_n) \leq w_d \leq w$, Then, we can find at most $2w/w_d$ vertical lines such that the sum of the weights of points strictly between (we do not count the points on these lines) any two adjacent lines is at most w_d in time $O(n \log(w/w_d))$.*

Algorithm 2 Partition($\{x_1, x_2, ..., x_n\}$)

Find the weighted median x_k (w.r.t. w-weight);
$\mathcal{L} = \mathcal{L} \cup \{x_k\}$;
Generate $S = \{x_i \mid w_i < x_k\}$, $L = \{x_i \mid w_i > x_k\}$;
If the sum of the weights of the points in S is lager than w_d, run Partition(S);
If the sum of the weights of the points in L is lager than w_d, run Partition(L);

Proof. See Algorithm 2. In this algorithm, we apply the weighted median algorithm recursively. Initially we have a global variable $\mathcal{L} = \emptyset$, which upon termination is the set of x-coordinates of the selected vertical lines. Each time we find the weighted median x_k and separate the point with the vertical line

$x = x_k$, which we add into $\mathcal{L}$. The sum of the weights of points in either side is at most half of the sum of the weights of all the points. Hence, the depth of the recursion is at most $\lceil \log(w/w_d) \rceil$. Thus, the size of $\mathcal{L}$ is at most $2^{\lceil \log(w/w_d) \rceil} \leq 2w/w_d$, and the running time is $O(n \log(w/w_d))$. $\square$

We describe how to partition cell c into smaller cells. First, we partition c with some vertical lines. Let $\mathcal{L}_v$ denote a set of vertical lines. Initially, $\mathcal{L}_v = \emptyset$. Let $w_d = \frac{\varepsilon \cdot W_\mathsf{c}}{16}$. We find all the points whose weights are at least w_d. For each such point, the vertical line that passes through this point is added to $\mathcal{L}_v$. Then, we apply Algorithm 2 to all the points with weights less than w_d. Next, we add a set $\mathcal{L}_h$ of horizontal lines in exactly the same way.

Lemma 2. *The sum of the weights of points strictly between any two adjacent lines in $\mathcal{L}_v$ is at most $w_d = \frac{\varepsilon \cdot W_\mathsf{c}}{16}$. The number of vertical lines in $\mathcal{L}_v$ is at most $\frac{32}{\varepsilon}$. Both statements hold for $\mathcal{L}_h$ as well.*

Proof. The first statement is straightforward from the description of the algorithm. We only need to prove the upper bound of the number of the vertical lines. Assume the sum of the weights of those points considered in the first (resp. second) step is W_1(resp. W_2), $W_1 + W_2 = W_\mathsf{c}$. The number of vertical lines in $\mathcal{L}_v$ is at most

$$W_1 / \left(\frac{\varepsilon \cdot W_\mathsf{c}}{16} \right) + 2W_2 / \left(\frac{\varepsilon \cdot W_\mathsf{c}}{16} \right) \leq \frac{32}{\varepsilon}.$$

The first term is due to the fact that the weight of each point we found in the first step has weight at least $\frac{\varepsilon \cdot W_\mathsf{c}}{16}$, and the second term directly follows from Lemma 1. $\square$

We add both vertical boundaries of cell c into $\mathcal{L}_v$ and both horizontal boundaries of cell c into $\mathcal{L}_h$. Now $\mathcal{L} = \mathcal{L}_v \cup \mathcal{L}_h$ forms a grid of size at most $(\frac{32}{\varepsilon}+2) \times (\frac{32}{\varepsilon}+2)$. Assume $\mathcal{L} = \{(x, y) \in \mathbb{R}^2 \mid y = y_j, j \in \{1, ..., v\}\} \cup \{(x, y) \in \mathbb{R}^2 \mid x = x_i, i \in \{1, ..., u\}\}$, with both $\{y_i\}$ and $\{x_i\}$ are sorted. $\mathcal{L}$ partitions c into *small cells*. The final step of our algorithm is simply enumerating all the unit squares located at $(x_i, y_j), i \in \{1, ..., u\}, j \in \{1, ..., v\}$, and return the one with the maximum coverage. However, computing the coverage exactly for all these unit squares is expensive. Instead, we only calculate the weight of these unit square approximately as follows. For each unit square r, we only count the weight of points that are in some small cell fully covered by r. Now, we show this can be done in $O\left(n_\mathsf{c} \log\left(\frac{1}{\varepsilon}\right) + \left(\frac{1}{\varepsilon}\right)^2\right)$ time.

After sorting $\{y_i\}$ and $\{x_i\}$, we can use binary search to identify which small cell each point lies in. So we can calculate the sum of the weights of points at the interior, edges or corners of all small cells in $O(n_{\mathsf{c}} \log\left(\frac{1}{\varepsilon}\right))$ time.

Thus searching the unit square with the maximum (approximate) coverage can be done with a standard incremental algorithm in $O\left(\frac{1}{\varepsilon}\right)^2$ time.

Putting everything together, we conclude that if $n_{\mathsf{c}} \geq \left(\frac{1}{\varepsilon}\right)^2$, the running time of MaxCovCell(c) is $O\left(n_c \log\left(\frac{1}{\varepsilon}\right) + \left(\frac{1}{\varepsilon}\right)^2\right)$.

Lemma 3. *The subroutine* MaxCovCell(c) *returns a (1-ε)-approximation to* $\mathsf{MaxCov_R}(\mathcal{P}_{\mathsf{c}}, 1)$*, where* $\mathcal{P}_{\mathsf{c}}$ *is the set of points in* $\mathcal{P}$ *that lies in* c.

Proof. The case $n_{\mathsf{c}} < \left(\frac{1}{\varepsilon}\right)^2$ is trivial since we apply the exact algorithm. So we only need to prove the case of $n_{\mathsf{c}} \geq \left(\frac{1}{\varepsilon}\right)^2$.

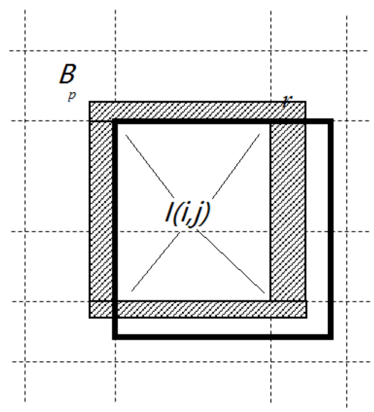

Figure 1: Proof of Lemma 3.

Suppose the optimal unit square is r. Denote by Opt the weight of the optimal solution. The size of c is 2×2, so we can use 4 unit squares to cover the entire cell. Therefore, $\mathsf{Opt} \geq \frac{W_{\mathsf{c}}}{4}$. Suppose r is located at a point p, which is in the strict interior of a small cell B separated by $\mathcal{L}$. [10] Suppose the index of B is (i, j). We compare the weight of r with $I(i, j)$ (which is the approximate weight of the unit square located at the top-left corner of B). See Figure 1. By the rule of our partition, the weight difference is at most 4 times the maximum possible weight of points between two adjacent

[10] If p lies on the boundary of B, the same argument still works.

lines in $\mathcal{L}$. Formally, $I(i,j) \geq \mathsf{Opt} - 4 \cdot \frac{\varepsilon \cdot W_{\mathsf{c}}}{16}$. Applying $\mathsf{Opt} \geq \frac{W_{\mathsf{c}}}{4}$, we get $I(i,j) \geq (1-\varepsilon)\mathsf{Opt}$. This proves the approximation guarantee. □

We conclude the main result of this section with the following theorem.

Theorem 1. *Algorithm 1 returns a (1-ε)-approximation to* $\mathsf{MaxCov}_{\mathsf{R}}(\mathcal{P}, 1)$ *in* $O(n \log\left(\frac{1}{\varepsilon}\right))$ *time.*

Proof. The correctness follows from Lemma 3 and the previous discussion. The running time consists of two parts: cells with number of points more than $\left(\frac{1}{\varepsilon}\right)^2$ and cells with number of points less than $\left(\frac{1}{\varepsilon}\right)^2$. Let $n_1 \geq ... \geq n_j \geq \left(\frac{1}{\varepsilon}\right)^2 > n_{j+1} \geq ... \geq n_{j+k}$ be the sorted sequence of the number of points in all cells. We have that

$$\begin{aligned}
\text{Running time} \leq & \sum_{i=1}^{j} O\left(n_i \log\left(\frac{1}{\varepsilon}\right) + \left(\frac{1}{\varepsilon}\right)^2\right) + \sum_{i=1}^{k} O\left(n_{i+j}\log(n_{i+j})\right) \\
=& O\left(\log\left(\frac{1}{\varepsilon}\right)\sum_{i=1}^{j} n_i + j\left(\frac{1}{\varepsilon}\right)^2 + \sum_{i=1}^{k} n_{i+j}\log(n_{i+j})\right) \\
\leq & O\left(\log\left(\frac{1}{\varepsilon}\right)\sum_{i=1}^{j}(n_i) + n + \sum_{i=1}^{k} n_{i+j}\log\left(\frac{1}{\varepsilon}\right)\right) \\
=& O\left(\log\left(\frac{1}{\varepsilon}\right)\sum_{i=1}^{j+k}(n_i) + n\right) = O\left(n\log\left(\frac{1}{\varepsilon}\right)\right).
\end{aligned}$$

Notice that we apply $j\left(\frac{1}{\varepsilon}\right)^2 = \underbrace{(1/\varepsilon)^2 + \ldots + (1/\varepsilon)^2}_{j} \leq n_1 + \ldots + n_j \leq n$. □

4. Linear Time Algorithms for $\mathsf{MaxCov}_{\mathsf{R}}(\mathcal{P}, m)$

For general m, we need the shifting technique [10].

4.1. Grid Shifting

Consider grids with a different side length $\frac{6}{\varepsilon}$. We shift the grid to $\frac{6}{\varepsilon}$ different positions: $(0,0),(1,1),....,(\frac{6}{\varepsilon}-1,\frac{6}{\varepsilon}-1)$. (For simplicity, we assume that $\frac{1}{\varepsilon}$ is an integer and no point in $\mathcal{P}$ has an integer coordinate, so points in $\mathcal{P}$ will never lie on the grid line. Let

$$\mathbb{G} = \left\{G_{6/\varepsilon}(0,0), ..., G_{6/\varepsilon}(6/\varepsilon - 1, 6/\varepsilon - 1)\right\}.$$

The following lemma is quite standard.

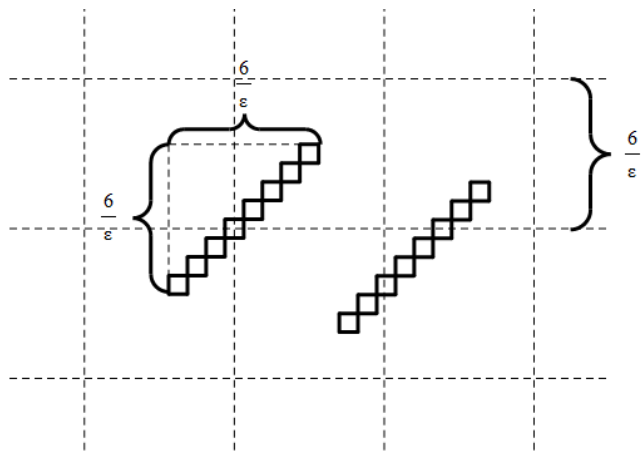

Figure 2: Proof of Lemma 4: the shifting technique.

Lemma 4. *There exist $G^\star \in \mathbb{G}$ and a $(1 - \frac{2\varepsilon}{3})$-approximate solution R such that none of the unit squares in R intersects $G^\star$.*

Proof. For any point p, we can always use four unit squares to cover the 2×2 square centered at p. Therefore, there exists an optimal solution OPT such that each covered point is covered by at most 4 unit squares in OPT.

For each grid $G_{\frac{6}{\varepsilon}}(i,i) \in \mathbb{G}$, we build a modified answer R_i from OPT in the following way. For each square r that intersects with $G_{\frac{6}{\varepsilon}}(i,i)$, there are two different situations. If r only intersects with one vertical line or one horizontal line. We move the square to one side of the line with bigger weight. In this case we will lose at most half of the weight of r. Notice that this kind of squares can only intersect with two grids in $\mathbb{G}$. Similarly, If r intersects with one vertical line and one horizontal line at the same time, we move it to one of the four quadrants derived by these two lines. In this case we will lose at most 3/4 of the weight of r. This kind of squares can only intersect with one grid in $\mathbb{G}$ (see Figure 2). Now we calculate the sum of the weights we lose from $R_0, R_1, ..., R_{\frac{6}{\varepsilon}-1}$, which is at most $\max\{1/2 \times 2, 3/4 \times 1\} = 1$ times the sum of weights of squares in OPT, which is at most $4w(\mathsf{OPT})$ due to the definition of OPT (which says that each covered point is covered by at most 4 unit squares in OPT). So the sum of the weights of $R_0, R_1, ..., R_{\frac{6}{\varepsilon}-1}$ is at least $(\frac{6}{\varepsilon} - 4)w(\mathsf{OPT})$. Therefore there exists some i such that R_i(which does not intersect $G_{\frac{6}{\varepsilon}}(i,i)$) is a $(1 - \frac{2\varepsilon}{3})$ approximate answer. $\square$

We will approximately solve the problem for each grid G in $\mathbb{G}$ (that is, find an approximation to R_G, where R_G denotes the best solution where no squares in R_G intersect G), and then select the optimal solution among them.

The idea to solve a fixed grid is as follows. First, we present a subroutine in Subsection 4.4 which can approximately solve the problem for a fixed cell. Then, we apply it to all the nonempty cells. To compute our final output from those obtained solutions, we apply a dynamic programming algorithm or a greedy algorithm which are shown in the next two sections.

Some common notation of the following two subsections are as follows. Assume $G \in \mathbb{G}$. Let $\mathsf{c}_1, \ldots, \mathsf{c}_t$ be the nonempty cells of grid G and Opt be the optimal solution that does not intersect G. Let $\mathsf{W}(\mathsf{c}_i, k)$ be the maximum weight we can cover with k unit squares in cell c_i. Let $m_c = \min\{m, (\frac{6}{\varepsilon})^2\}$. Because $(\frac{6}{\varepsilon})^2$ unit squares are enough to cover an entire $\frac{6}{\varepsilon} \times \frac{6}{\varepsilon}$ cell, the maximum number of unit squares we need to place in one single cell is m_c.

4.2. Dynamic Programming

For each nonempty cell c_i and for each $k \in [m_c]$, we find a $(1 - \frac{\varepsilon}{3})$-approximation $\mathsf{F}(\mathsf{c}_i, k)$ to $\mathsf{W}(\mathsf{c}_i, k)$. We will show how to achieve this later in Subsection 4.4. For now, assume that we can do it.

Let Opt_F be the optimal solution we can get from the values $\mathsf{F}(\mathsf{c}_i, k)$. More precisely,

$$\mathsf{Opt}_\mathsf{F} = \max_{k_1,\ldots,k_t \in [m_c]} \left\{ \sum_{i=1}^{t} \mathsf{F}(\mathsf{c}_i, k_i) \;\middle|\; \sum_{i=1}^{t} k_i = m \right\}. \tag{1}$$

We can see that Opt_F must be a $(1 - \frac{\varepsilon}{3})$-approximation to Opt. We can easily use dynamic programming to calculate the exact value of Opt_F. Denote by $A(i, k)$ the maximum weight we can cover with k unit squares in cells $\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_i$. We have the following DP recursion:

$$A(i,k) = \begin{cases} \max_{j=0}^{\min(k,m_c)} \{A(i-1, k-j) + \mathsf{F}(\mathsf{c}_i, j)\} & \text{if} \quad i > 1 \\ \mathsf{F}(c_1, k) & \text{if} \quad i = 1 \end{cases}$$

The running time of the above simple dynamic programming is $O(m \cdot t \cdot m_c)$. One may notice that each step of the DP is computing a $(+, \max)$ convolution. However, existing algorithms (see e.g., [27, 28]) only run slightly better than quadratic time. So the improvement would be quite marginal. But in the next section, we show that if we would like to settle for an approximation to Opt_F, the running time can be dramatically improved to linear.

4.3. A Greedy Algorithm

We first apply our $\mathsf{MaxCov_R}(\mathcal{P}, 1)$ algorithm in Section 3 to each cell c_i, to compute a $(1 - \frac{\varepsilon^2}{9})$-approximation of $\mathsf{W}(\mathsf{c}_i, 1)$. Let $f(\mathsf{c}_i, 1)$ be the return values. [11] This takes $O(n \log \frac{1}{\varepsilon})$ time. Then, we use the selection algorithm to find out the m cells with the largest $f(\mathsf{c}_i, 1)$ values. Assume that those cells are $\mathsf{c}_1, ..., \mathsf{c}_m, \mathsf{c}_{m+1}, ..., \mathsf{c}_t$, sorted from largest to smallest by $f(\mathsf{c}_i, 1)$.

Lemma 5. *Let $\mathsf{Opt}(m)$ be the maximum weight we can cover using m unit squares in $\mathsf{c}_1, ..., \mathsf{c}_m$. Then $\mathsf{Opt}(m) \geq (1 - \frac{\varepsilon^2}{9})\mathsf{Opt}$. (Note: Opt is the optimal solution that does not intersect G; not the globally optimum solution.)*

Proof. Let k be the number of unit squares in Opt that are contained in $\mathsf{c}_{m+1}, \ldots, \mathsf{c}_t$. This means there must be at least k cells in $\{\mathsf{c}_1, \ldots, \mathsf{c}_m\}$ such that Opt does not place any unit square. Therefore we can always move all k unit squares placed in $\mathsf{c}_{m+1}, \ldots, \mathsf{c}_t$ to these empty cells such that each empty cell contains only one unit square. Denote the weight of this modified solution by A. Obviously, $\mathsf{Opt}(m) \geq A$. For any i,j such that $1 \leq i \leq m < j \leq t$, we have $\mathsf{W}(\mathsf{c}_i, 1) \geq f(\mathsf{c}_i, 1) \geq f(\mathsf{c}_j, 1) \geq (1 - \frac{\varepsilon^2}{9})\mathsf{W}(\mathsf{c}_j, 1)$. Combining with a simple observation that $\mathsf{W}(\mathsf{c}_j, k) \leq k\mathsf{W}(\mathsf{c}_j, 1)$, we can see that $A \geq (1 - \frac{\varepsilon^2}{9})\mathsf{Opt}$. Therefore, $\mathsf{Opt}(m) \geq (1 - \frac{\varepsilon^2}{9})\mathsf{Opt}$. $\square$

Hence, from now on, we only need to consider the first m cells $\{\mathsf{c}_1, ..., \mathsf{c}_m\}$.

Recall that $\mathsf{F}(\mathsf{c}_i, k)$ denotes a $(1 - \frac{\varepsilon}{3})$-approximation to $\mathsf{W}(\mathsf{c}_i, k)$. Let $\mathsf{Opt_F}(m)$ be the optimal solution we can get from the values $\mathsf{F}(\mathsf{c}_i, k)$ of the first m cells. More precisely,

$$\mathsf{Opt_F}(m) = \max_{k_1,\ldots,k_m \in [m_c]} \left\{ \sum_{i=1}^{m} \mathsf{F}(\mathsf{c}_i, k_i) \;\middle|\; \sum_{i=1}^{m} k_i = m \right\}. \tag{2}$$

We distinguish two cases. If $m \leq 324(\frac{1}{\varepsilon})^4$, we just apply the dynamic program to compute $\mathsf{Opt_F}(m)$. The running time of the above dynamic programming is $O((\frac{1}{\varepsilon})^{O(1)})$. If $m > 324(\frac{1}{\varepsilon})^4$, we can use a greedy algorithm to find an answer of weight at least $(1 - \frac{\varepsilon^2}{9})\mathsf{Opt_F}(m)$.

Let $\mathsf{b} = (\frac{6}{\varepsilon})^2$. Notice that b unit squares are enough to cover an entire $\frac{6}{\varepsilon} \times \frac{6}{\varepsilon}$ cell. For each cell c_i, we find the upper convex hull of 2D points

[11] Both $f(\mathsf{c}_i, 1)$ and $\mathsf{F}(\mathsf{c}_i, 1)$ are approximations of $\mathsf{W}(\mathsf{c}_i, 1)$, with slightly different approximation ratios.

$\{(0, \mathsf{F}(\mathsf{c}_i, 0)),(1, \mathsf{F}(\mathsf{c}_i, 1)), \ldots, (\mathsf{b}, \mathsf{F}(\mathsf{c}_i, \mathsf{b}))\}$. See Figure 3. Suppose the convex hull points are $\{(t_{i,0}, \mathsf{F}(\mathsf{c}_i, t_{i,0})), (t_{i,1}, \mathsf{F}(\mathsf{c}_i, t_{i,1})), \ldots, (t_{i,s_i}, \mathsf{F}(\mathsf{c}_i, t_{i,s_i}))\}$, where $t_{i,0} = 0, t_{i,s_i} = \mathsf{b}$. For each cell, since the above points are already sorted from left to right, we can compute the convex hull in $O(\mathsf{b})$ time by Graham's scan[29]. Therefore, computing the convex hulls for all these cells takes $O(m\mathsf{b})$ time.

For each cell c_i, we maintain a value p_i representing that we are going to place t_{i,p_i} squares in cell c_i. Initially for all $i \in [m]$, $p_i = 0$. In each stage, we find the cell c_i such that current slope (the slope of the next convex hull edge)

$$\frac{\mathsf{F}(\mathsf{c}_i, t_{i,p_i+1}) - \mathsf{F}(\mathsf{c}_i, t_{i,p_i})}{t_{i,p_i+1} - t_{i,p_i}}$$

is maximized. Then we add 1 to p_i, or equivalently we assign $t_{i,p_i+1} - t_{i,p_i}$ more squares into cell c_i. We repeat this step until we have already placed at least $m - \mathsf{b}$ squares. We can always achieve this since we can place at most b squares in one single cell in each iteration. Let m' the number of squares we have placed ($m - \mathsf{b} \le m' \le m$). For the remaining $m - m'$ squares, we allocate them arbitrarily. We denote the algorithm by GREEDY and let the value obtained be $\mathsf{Greedy}(m')$. Having the convex hulls, the running time of the greedy algorithm is $O(m \log m)$.

Now we analyze the performance of the greedy algorithm.

Lemma 6. *The above greedy algorithm computes an $(1-\varepsilon^2/9)$-approximation to $\mathsf{Opt}_\mathsf{F}(m)$.*

Proof. Define an auxiliary function $\widehat{\mathsf{F}}(\mathsf{c}_i, k)$ as follows: If $k = t_{i,j}$ for some j, $\widehat{\mathsf{F}}(\mathsf{c}_i, k) = \mathsf{F}(\mathsf{c}_i, k)$. Otherwise, suppose $t_{i,j} < k < t_{i,j+1}$, then

$$\widehat{\mathsf{F}}(\mathsf{c}_i, k) = \mathsf{F}(\mathsf{c}_i, t_{i,j}) + \frac{\mathsf{F}(\mathsf{c}_i, t_{i,j+1}) - \mathsf{F}(\mathsf{c}_i, t_{i,j})}{t_{i,j+1} - t_{i,j}} \times (k - t_{i,j}).$$

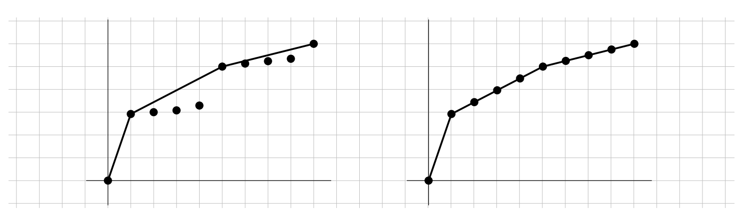

Figure 3: $\mathsf{F}(\mathsf{c}_i, k)$ (left) and $\widehat{\mathsf{F}}(\mathsf{c}_i, k)$ (right)

Intuitively speaking, $\widehat{\mathsf{F}}(\mathsf{c}_i, k)$(See Figure 3) is the function defined by the upper convex hull at integer points. [12] Thus, for all $i \in [m]$, $\widehat{\mathsf{F}}(\mathsf{c}_i, k)$ is a concave function. Obviously, $\widehat{\mathsf{F}}(\mathsf{c}_i, k) \geq \mathsf{F}(\mathsf{c}_i, k)$ for all $i \in [m]$ and all $k \in [\mathsf{b}]$.

Let $\mathsf{Opt}_{\widehat{\mathsf{F}}}(i)$ be the optimal solution we can get from the values $\widehat{\mathsf{F}}(\mathsf{c}_i, k)$ by placing i squares. By the convexity of $\widehat{\mathsf{F}}(\mathsf{c}_i, k)$, the following greedy algorithm is optimal: as long as we still have budget, we assign 1 more square to the cell which provides the largest increment of the objective value. In fact, this greedy algorithm runs in almost the same way as Greedy. The only difference is that Greedy only picks an entire edge of the convex hull, while the greedy algorithm here may stop in the middle of an edge (this may only happen for the last edge). Since the marginal increment never increases, we can see that $\mathsf{Opt}_{\widehat{\mathsf{F}}}(i)$ is concave.

By the way of choosing cells in our greedy algorithm, we make the following simple but important observation:

$$\mathsf{Greedy}(m') = \mathsf{Opt}_{\widehat{\mathsf{F}}}(m') = \mathsf{Opt}_{\mathsf{F}}(m').$$

So, our greedy algorithm is in fact optimal for m'. Combining with $m - m' \leq \mathsf{b}$ and the concavity of $\mathsf{Opt}_{\widehat{\mathsf{F}}}$, we can see that

$$\mathsf{Opt}_{\widehat{\mathsf{F}}}(m') \geq \frac{m - \mathsf{b}}{m} \mathsf{Opt}_{\widehat{\mathsf{F}}}(m) \geq \left(1 - \frac{\varepsilon^2}{9}\right) \mathsf{Opt}_{\widehat{\mathsf{F}}}(m) \geq \left(1 - \frac{\varepsilon^2}{9}\right) \mathsf{Opt}_{\mathsf{F}}(m).$$

The last inequality holds because $\mathsf{Opt}_{\widehat{\mathsf{F}}}(i) \geq \mathsf{Opt}_{\mathsf{F}}(i)$ for any i. The second last inequality holds because $m > \frac{324}{\varepsilon^4}$ and $b = \frac{36}{\varepsilon^2}$. □

4.4. Computing $\mathsf{F}(\mathsf{c}, k)$

Now we show the subroutine MaxCovCellM for computing $\mathsf{F}(\mathsf{c}, k)$.

We use a similar partition algorithm as Section 3.2. The only difference is that this time we need to partition the cell finer so that the maximum possible weight of points between any two adjacent parallel partition lines is $(\frac{\varepsilon^3 W_{\mathsf{c}}}{432})$. After partitioning the cell, we enumerate all the possible ways of placing k unit squares at the grid point. Similarly, for each unit square r, we only count the weight of points that are in some cell fully covered by r.

[12] At first sight, it may appear that $\mathsf{F}(\mathsf{c}_i, k)$ should be a concave function. However, this is not true. A counter-example is provided in the appendix.

Enumeration in MaxCovCellM. We can adapt the algorithm in [9] to enumerate these possible ways of placing k unit squares at the grid point in $O((\frac{1}{\varepsilon})^{\Delta})$ time where $\Delta = O(\sqrt{k})$. We briefly sketch the algorithm here.

Denote the optimal solution as $\mathsf{Opt}_{\mathsf{c}}$. From [30] we know that for any optimal solution, there exists a line (either horizontal or vertical) in the grid that intersects with $O(\sqrt{k})$ squares in $\mathsf{Opt}_{\mathsf{c}}$, denoted as the *parting line*. So we can use dynamic programming. At each stage, we enumerate the parting line, and the $O(\sqrt{k})$ squares intersecting the parting line. We also enumerate the number of squares in each side of the parting line in the optimal solution. The total number of choices is $O((\frac{1}{\varepsilon})^{\Delta})$. Then, we can solve recursively for each side. In the recursion, we should consider a subproblem which is composed of a smaller rectangle, and an enumeration of $O(\sqrt{k})$ squares of the optimal solution intersecting the boundary of the rectangle and at most k squares fully contained in the rectangle. Overall, the dynamic programming can be carried out in $O((\frac{1}{\varepsilon})^{\Delta})$ time.

Now we prove the correctness of this algorithm.

Lemma 7. MaxCovCellM *returns a* $(1 - \frac{\varepsilon}{3})$ *approximation to* $\mathsf{W}(\mathsf{c}_i, k)$.

Proof. We can use $(\frac{6}{\varepsilon})^2$ unit squares to cover the entire cell, so $\mathsf{W}(\mathsf{c}_i, k) \geq \frac{k\varepsilon^2 W_{\mathsf{c}}}{36}$. By the same argument as in Theorem 1, the difference between $\mathsf{W}(\mathsf{c}_i, k)$ and the answer we got are at most $4k$ times the maximum possible weight of points between two adjacent parallel partition lines. Therefore, the algorithm returns a $(1 - \frac{\varepsilon}{3})$-approximate answer of $\mathsf{W}(\mathsf{c}_i, k)$. □

Now we can conclude the following theorem.

Theorem 2. *Let P be a set of n weighted point, for any $0 < \varepsilon < 1$ we can find a $(1-\varepsilon)$-approximate answer for* $\mathsf{MaxCov}_{\mathsf{R}}(\mathcal{P}, m)$ *in time*

$$O\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon} + \frac{m}{\varepsilon}\log m + m\left(\frac{1}{\varepsilon}\right)^{\Delta_1}\right),$$

where $\Delta_1 = O(\min(\sqrt{m}, \frac{1}{\varepsilon}))$.

Proof. The algorithm is summarized in Algorithm 3. By Lemma 6, the greedy algorithm computes a $(1 - \varepsilon^2/9)$-approximation to $\mathsf{Opt}_{\mathsf{F}}(m)$. Since $\mathsf{F}(\mathsf{c}_i, k)$ is $(1-\frac{\varepsilon}{3})$-approximation to $\mathsf{W}(\mathsf{c}_i, k)$, we get $\mathsf{Opt}_{\mathsf{F}}(m) \geq (1-\frac{\varepsilon}{3})\mathsf{Opt}(m)$.

Algorithm 3 $\mathsf{MaxCov_R}(\mathcal{P}, m)$

$w_{\max} \leftarrow 0$
for each $G \in \{G_{\frac{6}{\varepsilon}}(0,0), ..., G_{\frac{6}{\varepsilon}}(\frac{6}{\varepsilon}-1, \frac{6}{\varepsilon}-1)\}$ **do**
  Use perfect hashing to find all the non-empty cells of G.
  for each non-empty cell c of G **do**
    $r_{\mathsf{c}} \leftarrow$ Algorithm 1 for c with approximate ratio $(1-\frac{\varepsilon^2}{9})$
  end for;
  Find the m cells with the largest r_{c}. Suppose they are $\mathsf{c}_1, ..., \mathsf{c}_m$.
  for $i \leftarrow 1$ **to** m **do**
    for $k \leftarrow 1$ **to** b **do** $\mathsf{F}(\mathsf{c}_i, k) \leftarrow$ MaxCovCellM(c,k)
    end for
  end for;
  if $m \leq 324(\frac{1}{\varepsilon})^4$, **then** $r \leftarrow \mathrm{DP}(\{\mathsf{F}(\mathsf{c}_i, k)\})$
  else $r \leftarrow$ Greedy$(\{\mathsf{F}(\mathsf{c}_i, k)\})$
  if $w(r) > w_{\max}$, **then** $w_{\max} \leftarrow w(r)$ and $r_{\max} \leftarrow r$
end for;
return $r_{\max}$;

By Lemma 5, we get $\mathsf{Opt}(m) \geq (1-\frac{\varepsilon^2}{9})\mathsf{Opt}$. (Recall that Opt denotes the optimal solution that does not intersect G.) Altogether, the greedy algorithm computes an $(1-\varepsilon^2/9)(1-\varepsilon^2/9)(1-\varepsilon/3)$ approximation to Opt. Moreover, by Lemma 4, Algorithm 3 returns a $(1-\frac{2\varepsilon}{3})(1-\frac{\varepsilon^2}{9})(1-\frac{\varepsilon^2}{9})(1-\frac{\varepsilon}{3})$ approximation to the original problem. Since

$$(1-\frac{2\varepsilon}{3})(1-\frac{\varepsilon^2}{9})(1-\frac{\varepsilon^2}{9})(1-\frac{\varepsilon}{3}) > (1-\varepsilon),$$

Algorithm 3 does return a $(1-\varepsilon)$-approximate solution.

We now calculate the running time. Solving the values $f(\mathsf{c}_i, 1)$ and finding out the top m results require $O(n \log \frac{1}{\varepsilon})$ time. We compute the values $\mathsf{F}(\mathsf{c}_i, k)$ of m cells. For each cell c_i, we partition it only once and calculate $\mathsf{F}(\mathsf{c}_i, 1), \ldots, \mathsf{F}(\mathsf{c}_i, b)$ using the same partition. Computing the values $\mathsf{F}(\mathsf{c}_i, k)$ of all m cells requires $O(n\log(\frac{1}{\varepsilon})+m(\frac{1}{\varepsilon})^{\Delta_1})$ time. The greedy algorithm costs $O(m \log m)$ time. We do the same for $\frac{6}{\varepsilon}$ different grids. Therefore, the overall running time is as we state in the theorem. □

5. Extension to Other Shapes

Our algorithm can easily be extended to solve other shapes. We show the extension in this section. The framework is almost the same as before. The major difference is the way for building an $(1-\varepsilon)$-approximation in each cell (the partition scheme in Section 4.4 works only for rectangles).

5.1. Assumptions on the general shape

Now, we assume that D is a shape subject to the following conditions.

- C-1 It is connected and closed, and its boundary is a simple closed curve.
- C-2 It is contained in an axis-parallel square of size 1×1, and on the other hand it contains an axis-parallel square of size $\sigma \times \sigma$, where $\sigma = \Omega(1)$. For convenience, we assume that $\frac{1}{\sigma}$ is an integer.
- C-3 Let $\partial\mathsf{D}$ denote the boundary of D. If we place k copies of D in $\mathbb{R}^2$, the arrangement defined by their boundaries contains at most $O(k^2)$ cells.

Remark: The above assumptions are quite general. Now, we list some shapes satisfying those assumptions.

1. Disks and ellipsoid;
2. Convex polygons with constant size (e.g. triangles). For a convex body C in the plane, Lassak [31] showed that there is a rectangle r inscribed in C such that a homothetic copy R of r is circumscribed about C and the homothetic ratio is at most 2. Therefore, we can always affine-transform a convex body so that it satisfies C-2, with $\sigma = 1/2$. C-3 is also easy to see: in the arrangement defined by their boundaries, there are $O(k^2)$ intersection points or segments. Since the arrangement defines a planar graph, by Eular's formula, there are $O(k^2)$ cells.
3. Following the same argument, we can also handle the case where D satisfies C-1 and C-2, and the boundary of D comprises of τ bounded degree arcs, where τ is a fixed constant. By "bounded degree", we mean that there exists a constant deg so that each arc on the boundary of D is a polynomial curve with degree less than or equal to deg.

5.2. The shifting technique

For the general shape, we consider grids with side length $s = 6/(\sigma^2\varepsilon)$.

Again for simplicity, we assume that $\frac{1}{\varepsilon}$ is an integer and no point in $\mathcal{P}$ has an integer coordinate. We shift the grid to s different positions: $(0,0),(1,1),....,(s-1,s-1)$. Let $\mathbb{G} = \{G_s(0,0),...,G_s(s-1,s-1)\}$.

As we will see in the next lemma, the description of the shifting technique will be slightly more complicated than the original case for the squares. In the original case, for each grid G in $\mathbb{G}$ we shift the m squares so that no squares intersect with G. In the general case, we do **not** shift the shapes. Instead, for each grid G, we "assign" each of the m copies of D into one cell of G. By assigning a copy to a cell c, we do not shift it to make it lie in c (so, we do not require that this copy lies entirely inside c; it may intersect the boundary of c and so intersects G). When a copy D' is assigned to cell c of G, we assume that it only covers the points inside c. The *effective region* of D' is defined as $\mathsf{D}' \cap \mathsf{c}$.

Lemma 8. *There exist $G^\star \in \mathbb{G}$ such that we can place m copies of D and assign these copies to the cells of $G^\star$, so that the union of effective regions of these copies covers $(1 - \frac{2}{3}\varepsilon) \times \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}, m)$ weight of points.*

An equivalent description is the following.

Lemma 9. *For a grid G in $\mathbb{G}$, let $\mathsf{c}_1, \ldots, \mathsf{c}_t$ denote the nonempty cells. Define*

$$\mathsf{Opt}_G = \max \left(\sum_i \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}_i}, k_i) \mid \sum_i k_i = m \right).$$

Then, $(\max_{G \in \mathbb{G}} \mathsf{Opt}_G)$ is a $(1 - \frac{2}{3}\varepsilon)$-approximation of $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}, m)$.

Proof of Lemma 8. The proof is similar to that of Lemma 4.

For any point p, we can always use $(2/\sigma)^2$ copies of D to cover the 2×2 square centered at p. Therefore, there exists an optimal solution OPT such that each covered point is covered by at most $(2/\sigma)^2$ copies in OPT.

For each grid $G_s(i, i) \in \mathbb{G}$, we build a modified answer R_i from OPT in the following way. For each copy D' of D that intersects with $G_s(i, i)$, there are two different situations. If D' only intersects with one vertical line or one horizontal line. We assign D' to one side of the line with bigger weight. In this case we will lose at most half of the weight of D'. Notice that this kind of copies can only intersect with two grids in $\mathbb{G}$. Similarly, If D' intersects with one vertical line and one horizontal line at the same time, we assign it to one of the four quadrants derived by these two lines to keep the most weight. In this case we will lose at most $3/4$ of the weight of D'. This kind of copies can only intersect with one grid in $\mathbb{G}$. Now we calculate the sum of the weights we lose from $R_0, R_1, ..., R_{s-1}$, which is at most $\max\{1/2 \times 2, 3/4 \times 1\} = 1$

times the sum of weights of copies in OPT, which is at most $(2/\sigma)^2 w(\mathsf{OPT})$ due to the definition of OPT (which says that each covered point is covered by at most $(2/\sigma)^2$ copies in OPT). So the sum of the "effective weights" of $R_0, R_1, ..., R_{s-1}$ is at least $(s - (2/\sigma)^2) \cdot w(\mathsf{OPT})$. The effective weight of R_i is defined as the total weight covered by the union of the effective regions of R_i. Recall that $s = 6/(\sigma^2\varepsilon)$. By the pigeon-hole principle, there exists some i such that the effective weight of R_i is at least $(1 - \frac{2\varepsilon}{3}) \cdot w(\mathsf{OPT})$. □

5.3. Compute a $(1-\varepsilon)$*-approximation to* $\mathsf{MaxCov_D}(\mathcal{P}, m)$

We give the framework in Algorithm 4.

Algorithm 4 $\mathsf{MaxCov_D}(\mathcal{P}, m)$

$w_{\max} \leftarrow 0$
for each $G \in \mathbb{G}$ **do**
  Use perfect hashing to find all the non-empty cells of G.
  for each non-empty cell c of G
    $v_{\mathsf{c}} \leftarrow (1 - \frac{\varepsilon^2}{9})$ approximation to $\mathsf{MaxCov_D}(\mathcal{P}_{\mathsf{c}}, 1)$
  Find the m cells with the largest v_{c}. Suppose they are $\mathsf{c}_1, ..., \mathsf{c}_m$.
  Let $b \leftarrow \min(m, \frac{s^2}{\sigma^2}) = \min(m, \frac{36}{\sigma^6\varepsilon^2})$, which is the maximum number of copies put into a cell.
  for $i \leftarrow 1$ **to** m **do**
    Let c denote c_i for short.
    for $k \leftarrow 1$ **to** b
      $\mathsf{F}(\mathsf{c}_i, k) \leftarrow (1 - \frac{\varepsilon}{3})$ approximation to $\mathsf{MaxCov_D}(\mathcal{P}_{\mathsf{c}}, k)$
  end for;
  if $m \leq \frac{324}{\sigma^6\varepsilon^4}$, ($\frac{324}{\sigma^6\epsilon^4}$ is chosen so that $\frac{m-b}{m} \geq (1 - \varepsilon^2/9)$ and Lemma 6 remains true.) **then** $r \leftarrow \mathrm{DP}(\{\mathsf{F}(\mathsf{c}_i, k)\})$ **else** $r \leftarrow \textsc{Greedy}(\{\mathsf{F}(\mathsf{c}_i, k)\})$
  if $w(r) > w_{\max}$, **then** $w_{\max} \leftarrow w(r)$ and $r_{\max} \leftarrow r$
end for;
return $r_{\max}$;

The correctness proof is exactly the same as the proof for Algorithm 3. (Note that variable b here has a value different from that in Algorithm 3.)

Although, the framework is the same, the way for computing approximation of $\mathsf{MaxCov_D}(\mathcal{P}_{\mathsf{c}}, 1)$ and $\mathsf{MaxCov_D}(\mathcal{P}_{\mathsf{c}}, k)$ is different from the square case, since the partition technique does not apply. We show the new method in the next subsection and then analyze the running time of Algorithm 4.

5.4. Compute a $(1-\varepsilon)$*-approximation to* $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k)$

Definition. For a weighted point set $\mathcal{P}$ and a range space $\mathcal{U}$ (which is a set of regions in the plane), we say another weighted point set $\mathcal{A}$ is a $1/r$-approximation of $\mathcal{P}$ with respect to $\mathcal{U}$, if $\mathcal{A}$ and $\mathcal{P}$ have the same total weights and $|w(\mathcal{A} \cap U) - w(\mathcal{P} \cap U)| < w(\mathcal{P})/r$ for any $u \in \mathcal{U}$.

Let $\mathcal{U}_b$ be the collection of sets that are the union of b copies of D. Let $\mathcal{U}_1 = \{S \mid S \text{ is a translate of } \mathsf{D}\}$. Let $r_\varepsilon = 72/(\varepsilon^3\sigma^6)$ and denote it by r when ε is clear. The following lemma follows a very similar argument in [9].

Lemma 10. *Assume that* $\mathcal{A}_{\mathsf{c}}$ *is a* $1/r_\varepsilon$*-approximation of* $\mathcal{P}_{\mathsf{c}}$ *with respect to* $\mathcal{U}_b$*. For* $1 \le k \le b$*, if* $U^\star_{\mathcal{A}}$ *is an optimal solution for* $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{A}_{\mathsf{c}}, k)$*, then it is an* $(1-\varepsilon)$*-approximation to* $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k)$*.*

Proof. Let $U^\star_{\mathcal{P}}$ denote the optimal solution for $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k)$. Since $U^\star_{\mathcal{A}}$ is optimal for $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{A}_{\mathsf{c}}, k)$, we have $w(U^\star_{\mathcal{A}} \cap \mathcal{A}_{\mathsf{c}}) \ge w(U^\star_{\mathcal{P}} \cap \mathcal{A}_{\mathsf{c}})$.

Since c is of size $s \times s$, and D contains an axis-parallel square with size $\sigma \times \sigma$, we have $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, 1) \ge \frac{\sigma^2}{s^2} w(\mathcal{P}_{\mathsf{c}})$. Recall that $s = \frac{6}{\sigma^2\varepsilon}$. So,

$$\frac{1}{r} w(\mathcal{P}_{\mathsf{c}}) \le \frac{s^2}{\sigma^2 r} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, 1) = \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, 1) \le \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k).$$

Since $\mathcal{A}_{\mathsf{c}}$ is a $1/r$-approximation of $\mathcal{P}_{\mathsf{c}}$, we have

$$|w(U^\star_{\mathcal{A}} \cap \mathcal{A}_{\mathsf{c}}) - w(U^\star_{\mathcal{A}} \cap \mathcal{P}_{\mathsf{c}})| \le w(\mathcal{P}_{\mathsf{c}})/r \le \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k)$$

and

$$|w(U^\star_{\mathcal{P}} \cap \mathcal{A}_{\mathsf{c}}) - w(U^\star_{\mathcal{P}} \cap \mathcal{P}_{\mathsf{c}})| \le w(\mathcal{P}_{\mathsf{c}})/r \le \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k).$$

Therefore

$$\begin{aligned}
w(U^\star_{\mathcal{A}} \cap \mathcal{P}_{\mathsf{c}}) &= w(U^\star_{\mathcal{A}} \cap \mathcal{A}_{\mathsf{c}}) - w(U^\star_{\mathcal{A}} \cap \mathcal{A}_{\mathsf{c}}) + w(U^\star_{\mathcal{A}} \cap \mathcal{P}_{\mathsf{c}}) \\
&\ge w(U^\star_{\mathcal{A}} \cap \mathcal{A}_{\mathsf{c}}) - \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k) \\
&\ge w(U^\star_{\mathcal{P}} \cap \mathcal{A}_{\mathsf{c}}) - \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k) \\
&= w(U^\star_{\mathcal{P}} \cap \mathcal{P}_{\mathsf{c}}) - w(U^\star_{\mathcal{P}} \cap \mathcal{P}_{\mathsf{c}}) + w(U^\star_{\mathcal{P}} \cap \mathcal{A}_{\mathsf{c}}) - \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k) \\
&\ge w(U^\star_{\mathcal{P}} \cap \mathcal{P}_{\mathsf{c}}) - \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k) - \frac{\varepsilon}{2} \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k) \\
&= (1-\varepsilon) \mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k)
\end{aligned}$$

This finishes the proof of the lemma. $\square$

By this lemma, to compute an $(1-\varepsilon)$ approximation of $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, k)$, we can first build a $(1/r_\varepsilon)$-approximation $\mathcal{A}_{\mathsf{c}}$ of $\mathcal{P}_{\mathsf{c}}$ with respect to $\mathcal{U}_b$ (for some $b \geq k$) and then apply an exact algorithm to $\mathcal{A}_{\mathsf{c}}$.

First, we show how to build a $1/r$-approximation $\mathcal{A}_{\mathsf{c}}$ of $\mathcal{P}_{\mathsf{c}}$. We assume the reader is familiar with $1/r$-approximation for general range spaces.

Definition. For a range space $(X,\mathcal{U})$ with shattering dimension d, we say that it admits a *subspace oracle*, if given a set $Y \subseteq X$, a list of all distinct sets of the form $Y \cap U$ for some $U \in \mathcal{U}$ can be returned in $O(|Y|^{d+1})$ time.

Lemma 11 ([32])**.** *Let X be a weighted point set. Assume $(X,\mathcal{U})$ is a range space with shattering dimension d and admits a subspace oracle. For any parameter r, we can deterministically compute a $1/r$-approximation of size $O(r^2 \log r)$ for X with respect to $\mathcal{U}$, in time $O(|X| \cdot (r^2 \log r)^d)$.*

Lemma 12. *Suppose that X is a set of weighted points and r is a real.*

(1) We can construct a $1/r$-approximation of X with respect to $\mathcal{U}_1$, of size $O(r^2 \log r)$, in $O(r^4 \log^2 r|X|)$ time.

(2) For an integer $b > 1$, we can construct a $1/r$-approximation of X with respect to $\mathcal{U}_b$, of size $O((rb^2)^2 \log(rb^2))$, in $O((rb^2)^{12} \log^6(rb^2)|X|)$ time.

Proof. (1) First of all, we claim that $(X,\mathcal{U}_1)$ has shattering dimension 2.

We designate a fixed special point in D, called the pivot point of D. Let RD denote the shape constructed by rotating D by π. Assume that the pivot point of RD corresponds to the pivot point of D.

For any point $A \in \mathbb{R}^2$, when we say "we place a copy of D (or RD) at A", it means the pivot point of the copy is placed at A.

Assume $X = \{X_1, \ldots, X_k\}$ is a set of k points in $\mathbb{R}^2$. For each point X_i, we place a copy of RD at X_i (denoted by RD_i). Be aware that if we place a copy of D such that its pivot point is in RD_i, this copy of D can cover X_i.

Let Γ denote the arrangement of the boundaries of these k copies of RD. By C-3, there are $O(k^2)$ cells in this arrangement. By the way we define these cells, a copy of D placed in any point of the same cell covers the same subset of X. Therefore, the number of different subsets of X that are shattered by $\mathcal{U}_1$ is bounded by the number of cells of Γ. Hence, $(X,\mathcal{U}_1)$ has shattering dimension 2.

Next, we define a superset $\mathcal{U}_1^*$ of $\mathcal{U}_1$ and then construct a $1/r$-approximation with respect to $\mathcal{U}_1^*$. Since $\mathcal{U}_1^*$ is a superset of $\mathcal{U}_1$, the approximation with respect to $\mathcal{U}_1^*$ is also a $1/r$-approximation with respect to $\mathcal{U}_1$, and thus we get

(1). The reason we need to introduce $\mathcal{U}_1^*$ is that it is much easier to construct a subspace oracle for $(X, \mathcal{U}_1^*)$ instead of $(X, \mathcal{U}_1)$.

Define $\mathcal{U}_1^*$ as the union of $\mathcal{U}_1$ and $\{x \mid x \in X\}$ (i.e. each single point in X constitutes a set in $\mathcal{U}_1^*$). Note that $(X, \mathcal{U}_1^*)$ has shattering dimension 2. This immediately follows from the fact that $(X, \mathcal{U}_1)$ has shattering dimension 2.

We now construct a subspace oracle for $(X, \mathcal{U}_1^*)$. Given a subset $Y \subseteq X$, we should return the sets in $\mathcal{S} = \{Y \cap U \mid U \in \mathcal{U}_1^*\}$ in $|Y|^3$ time. We make an assumption that the points in X are in general positions. This means:

i For each pair of points X_i, X_j, the intersection between $\partial\mathsf{RD}_i$ and $\partial\mathsf{RD}_j$ are **isolated** points - the boundaries do not share a piece of curve.

ii Any three boundaries $\partial\mathsf{RD}_i, \partial\mathsf{RD}_j, \partial\mathsf{RD}_k$ $(i \neq j \neq k)$ do not intersect at the same point.

The details of the oracle is as follows. First, output each single point in Y, since each of them forms a single-element set shattered by $\mathcal{U}_1^*$. Next, output other subsets in $\mathcal{S}$ as follows. For every pair of points y_1, y_2 in Y, enumerate all the copies of D whose boundary passes through y_1, y_2 (such a copy corresponds exactly to an intersection between $\partial\mathsf{RD}_i$ and $\partial\mathsf{RD}_j$ when $y_1 = X_i$ and $y_2 = X_j$). Then, compute the set S of points in Y contained in the current copy, and output four sets: $S, S - \{y_1\}, S - \{y_2\}, S - \{y_1, y_2\}$. The correctness of this algorithm is briefly illustrated in Figure 4.

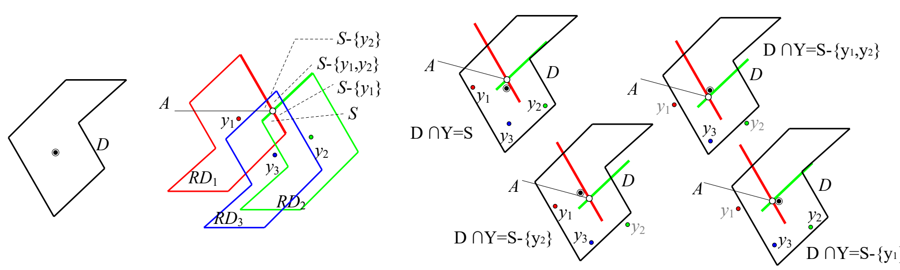

Figure 4: Illustration of the correctness. In the figure, A denotes an intersection of two segments that respectively comes from $\partial\mathsf{RD}_1$ and $\partial\mathsf{RD}_2$. These two segments divide the plane into four quadrants. Let S denote the subset of Y intersected by D when D is put on A. The pictures on the right show that if we put (pivot of) D in the four quadrants (and close to A), the subset in Y intersected by D would be $S, S - \{y_1\}, S - \{y_2\}, S - \{y_1, y_2\}$.

To sum up, $(X, \mathcal{U}_1^*)$ has shattering dimension 2 and admits a subspace oracle. Therefore, by applying Lemma 11, we can construct a $1/r$-approximation of X with respect to $\mathcal{U}_1^*$, of size $O(r^2 \log r)$, in $O(r^4 \log^2 r |X|)$ time.

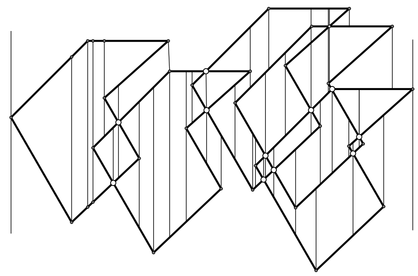

Figure 5: Vertical Decomposition.

(2) Let $\mathcal{V}$ be the infinite set of cells that can arise in a *vertical decomposition* of any collection of copies of D in the plane. (A vertical decomposition is the partition of the plane into interior disjoint vertical trapezoids formed by erecting vertical walls through each vertex. See Chapter 6 in [33] or Chapter 6 in [34] and Figure 5.) It is easy to see that $(X, \mathcal{V})$ has shattering dimension 6 (as proved in Lemma 3 in [9]). Briefly speaking, a cell is characterized by 6 points: the leftmost point, the rightmost point, two points on the upper boundary of this cell, and two points on the lower boundary of this cell. It follows that there is a subspace oracle for range space $(X, \mathcal{V})$.

Let $r' = v \cdot r$, where v is the maximum number of cells that the vertical decomposition of b copies of D can have. According to the assumption of D, given two different placements of D, their boundaries have constant many intersections. This implies that any arrangement of b copies of D has $O(b^2)$ cells, which further implies that $v = O(b^2)$. To construct a $1/r$-approximation for X with respect to $\mathcal{U}_b$, we apply the following fact proved in [9]: Let A be a $1/(r')$-approximation for X with respect to the ranges $\mathcal{V}$, then A is a $1/r$-approximation for X with respect to $\mathcal{U}_b$.

According to Lemma 11, we can construct a $1/(r')$-approximation for X of size $O(r'^2 \log r')$, in $O(((r')^2 \log(r'))^6 |X|)$ time. Thus we get (2). $\square$

Next, we show an exact algorithm for computing $\mathsf{MaxCov}_{\mathsf{D}}(X, m)$.

Lemma 13. *Assume that X is a set of weighted points. We can compute the exact solution to $\mathsf{MaxCov}_{\mathsf{D}}(X, m)$ in $O(|X|^{2m+1})$ time.*

Proof. In the optimal solution of $\mathsf{MaxCov}_{\mathsf{D}}(X, m)$, we can always choose those copies of D so that each of them contains a single point or contains two points on their boundary. We call these copies the critical copies. According to our assumption, the number of critical copies is $O(|X|^2)$, and these copies can be enumerated in $O(|X|^2)$ time. For each critical copy D_i, we compute and store in memory the list of points L_i that covered by D_i. Then, to compute the exact solution of $\mathsf{MaxCov}_{\mathsf{D}}(X, m)$, we enumerate all possible combination of m critical copies and find the optimum one. Since the points covered by each copy is stored, the points covered by the union of these copies can be computed in $O(|X|)$ time. Thus the running time is $O(|X|^{2m+1})$. [13] $\square$

5.5. Analysis of running time

Now we provide some details and the running time analysis of Algorithm 4. Recall that $r_\varepsilon = O(\varepsilon^{-3})$. For each non-empty cell c, we compute $\mathcal{A}_{\mathsf{c}}$, which is a $1/r_{\varepsilon^2/9}$-approximation of $\mathcal{P}_{\mathsf{c}}$ with respect to $\mathcal{U}_1$. This costs

$$O\left(n_c r^4_{\varepsilon^2/9} \log^2(r_{\varepsilon^2/9})\right) = O(n_c \varepsilon^{-25})$$

time according to Lemma 12 (1); and the size of $\mathcal{A}_{\mathsf{c}}$ is

$$r^2_{\varepsilon^2/9} \log(r_{\varepsilon^2/9}) = O(\varepsilon^{-13}).$$

Then, we compute $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{A}_{\mathsf{c}}, 1)$ and use it as the $(1-\varepsilon^2/9)$ approximation of $\mathsf{MaxCov}_{\mathsf{D}}(\mathcal{P}_{\mathsf{c}}, 1)$. This costs

$$O(|\mathcal{A}_{\mathsf{c}}|^3) = O(\varepsilon^{-39})$$

time according to Lemma 13. So the first inside loop costs $O(n\varepsilon^{-39})$ time.

For each cell c in $\mathsf{c}_1, \ldots, \mathsf{c}_m$, we compute a $1/r_{\varepsilon/3}$-approximation $\mathcal{A}'_{\mathsf{c}}$ of $\mathcal{P}_{\mathsf{c}}$ with respect to $\mathcal{U}_b$. Recall that $b = O(\varepsilon^{-2})$. The time for computing $\mathcal{A}'_{\mathsf{c}}$ is

$$O\left(n_c \times r^{12}_{\varepsilon/3} b^{24} \log^6(r_{\varepsilon/3} b^2)\right) = O(n_c \varepsilon^{-36-48-1}) = O(n_c (\frac{1}{\varepsilon})^{O(1)})$$

according to Lemma 12 (2). Therefore, the total time for computing $\mathcal{A}'_{\mathsf{c}}$ for $c \in \{\mathsf{c}_1, ..., \mathsf{c}_m\}$ is $O((\frac{1}{\varepsilon})^{O(1)} n)$. Note that $\mathcal{A}'_{\mathsf{c}}$ is of size

$$O\left(r^2_{\varepsilon/3} b^4 \log(r^2_{\varepsilon/3} b^4)\right) = O(\varepsilon^{-15}).$$

Therefore, the total time for computing $\{F(c_i, k) : 1 \le i \le m, 1 \le k \le b\}$ is $O(m(\varepsilon^{-15})^{2b+1})$ according to Lemma 13.

The overall running time is $O(n(\frac{1}{\varepsilon})^{O(1)} + \frac{m}{\epsilon} \log m + m(\frac{1}{\varepsilon})^{\Delta_2})$, where $\Delta_2 = O(\min(m, \frac{1}{\varepsilon^2}))$. The second term comes from the greedy algorithm.

[13] The running time can be improved to $O(|X|^2)$ time for $m = 1$. For $m = 1$, we do not need to store the list L_i for each i, and we can compute the summation of the weights in L_i in amortized $O(1)$ time. (See [15] or [9] for the details). We do not apply this optimization.

Reference

[1] D. S. Hochbaum, A. Pathria, Analysis of the greedy approach in problems of maximum k-coverage, Naval Research Logistics 45 (6) (1998) 615–627.

[2] G. L. Nemhauser, L. A. Wolsey, M. L. Fisher, An analysis of approximations for maximizing submodular set functions, Mathematical Programming 14 (1) (1978) 265–294.

[3] U. Feige, A threshold of ln n for approximating set cover, JACM 45 (4) (1998) 634–652.

[4] H. Imai, T. Asano, Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane, Journal of algorithms 4 (4) (1983) 310–323.

[5] S. C. Nandy, B. B. Bhattacharya, A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids, Computers & Mathematics with Applications 29 (8) (1995) 45–61.

[6] M. Ben-Or, Lower bounds for algebraic computation trees, in: STOC, ACM, 1983, pp. 80–86.

[7] Y. Tao, X. Hu, D.-W. Choi, C.-W. Chung, Approximate maxrs in spatial databases, Proceedings of the VLDB Endowment 6 (13) (2013) 1546–1557.

[8] N. Megiddo, K. J. Supowit, On the complexity of some common geometric location problems, SICOMP 13 (1) (1984) 182–196.

[9] M. de Berg, S. Cabello, S. Har-Peled, Covering many or few points with unit disks, Theory of Computing Systems 45 (3) (2009) 446–469.

[10] D. S. Hochbaum, W. Maass, Approximation schemes for covering and packing problems in image processing and vlsi, JACM 32 (1) (1985) 130–136.

[11] G. Barequet, M. Dickerson, P. Pau, Translating a convex polygon to contain a maximum number of points, Computational Geometry 8 (4) (1997) 167–179.

[12] M. Dickerson, D. Scharstein, Optimal placement of convex polygons to maximize point containment, Computational Geometry 11 (1) (1998) 1–16.

[13] Z. Drezner, Noteon a modified one-center model, Management Science 27 (7) (1981) 848–851.

[14] B. M. Chazelle, D.-T. Lee, On a circle placement problem, Computing 36 (1-2) (1986) 1–16.

[15] P. K. Agarwal, T. Hagerup, R. Ray, M. Sharir, M. Smid, E. Welzl, Translating a planar object to maximize point containment, in: ESA, Springer, 2002, pp. 42–53.

[16] B. Aronov, S. Har-Peled, On approximating the depth and related problems, SICOMP 38 (3) (2008) 899–921.

[17] S. Cabello, J. M. Díaz-Báñez, C. Seara, J. A. Sellares, J. Urrutia, I. Ventura, Covering point sets with two disjoint disks or squares, Computational Geometry 40 (3) (2008) 195–206.

[18] H. Brönnimann, M. Goodrich, Almost optimal set covers in finite vc-dimension, DCG 14 (1) (1995) 463–479. doi:10.1007/BF02570718.

[19] K. L. Clarkson, K. Varadarajan, Improved approximation algorithms for geometric set cover, DCG 37 (1) (2007) 43–58. doi:10.1007/s00454-006-1273-8.

[20] G. Even, D. Rawitz, S. M. Shahar, Hitting sets when the vc-dimension is small, IPL 95 (2) (2005) 358–362.

[21] N. H. Mustafa, S. Ray, Ptas for geometric hitting set problems via local search, in: SCG, ACM, 2009, pp. 17–22.

[22] K. Varadarajan, Weighted geometric set cover via quasi-uniform sampling, in: STOC, ACM, 2010, pp. 641–648.

[23] T. M. Chan, E. Grant, J. Könemann, M. Sharpe, Weighted capacitated, priority, and geometric set cover via improved quasi-uniform sampling, in: SODA, SODA '12, SIAM, 2012, pp. 1576–1585.

[24] J. Li, Y. Jin, A PTAS for the weighted unit disk cover problem, in: ICALP, Springer, 2015, pp. 898–909.

[25] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest-pair problem, Journal of Algorithms 25 (1) (1997) 19–51.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, The MIT Press, 2009.

[27] D. Bremner, T. M. Chan, E. D. Demaine, J. Erickson, F. Hurtado, J. Iacono, S. Langerman, P. Taslakian, Necklaces, convolutions, and x+y, in: ESA, Springer, 2006, pp. 160–171.

[28] R. Williams, Faster all-pairs shortest paths via circuit complexity, in: STOC, ACM, 2014, pp. 664–673.

[29] R. L. Graham, An efficient algorith for determining the convex hull of a finite planar set, Information processing letters 1 (4) (1972) 132–133.

[30] P. K. Agarwal, C. M. Procopiuc, Exact and approximation algorithms for clustering, Algorithmica 33 (2) (2002) 201–226.

[31] M. Lassak, Approximation of convex bodies by rectangles, Geometriae Dedicata 47 (1) (1993) 111–117.

[32] J. Matousek, Approximations and optimal geometric divide-and-conquer, Journal of Computer and System Sciences 50 (2) (1995) 203 – 208.

[33] S. Har-Peled, Geometric Approximation Algorithms, American Mathematical Society, Boston, MA, USA, 2011.

[34] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

Appendix A.

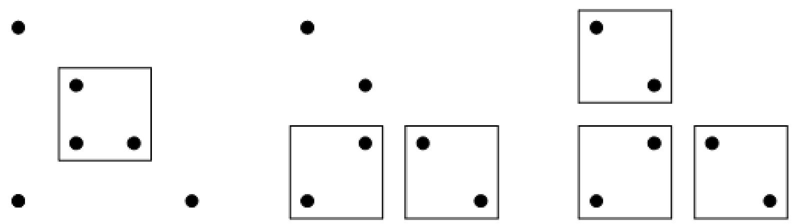

Figure A.6: $\mathsf{F}(\mathsf{c}_i, k)$ may not be concave: $\mathsf{F}(\mathsf{c}_i, 1) = 3$, $\mathsf{F}(\mathsf{c}_i, 2) = 4$, $\mathsf{F}(\mathsf{c}_i, 3) = 6$.