Range-clustering queries

Range-Clustering Queries*

Mikkel Abrahamsen[1], Mark de Berg[2], Kevin Buchin[2], Mehran Mehr[2], and Ali D. Mehrabi[2]

[1] Computer Science Department, University of Copenhagen, Denmark
[2] Computer Science Department, TU Eindhoven, the Netherlands

Abstract. In a geometric k-clustering problem the goal is to partition a set of points in $\mathbb{R}^d$ into k subsets such that a certain cost function of the clustering is minimized. We present data structures for orthogonal *range-clustering queries* on a point set S: given a query box Q and an integer $k \geqslant 2$, compute an optimal k-clustering for $S \cap Q$. We obtain the following results.

- We present a general method to compute a $(1+\varepsilon)$-approximation to a range-clustering query, where $\varepsilon > 0$ is a parameter that can be specified as part of the query. Our method applies to a large class of clustering problems, including k-center clustering in any L_p-metric and a variant of k-center clustering where the goal is to minimize the sum (instead of maximum) of the cluster sizes.
- We extend our method to deal with capacitated k-clustering problems, where each of the clusters should not contain more than a given number of points.
- For the special cases of rectilinear k-center clustering in $\mathbb{R}^1$, and in $\mathbb{R}^2$ for $k = 2$ or 3, we present data structures that answer range-clustering queries exactly.

1 Introduction

The range-searching problem is one of the most important and widely studied problems in computational geometry. In the standard setting one is given a set S of points in $\mathbb{R}^d$, and a query asks to report or count all points inside a geometric query range Q. In many applications, however, one would like to perform further analysis on the set $S \cap Q$, to obtain more information about its structure. Currently one then has to proceed as follows: first perform a range-reporting query to explicitly report $S \cap Q$, then apply a suitable analysis algorithm to $S \cap Q$. This two-stage process can be quite costly, because algorithms for analyzing geometric data sets can be slow and $S \cap Q$ can be large. To avoid this we would need data structures for what we call *range-analysis queries*, which directly compute the desired structural information about $S \cap Q$. In this paper we develop such data structures for the case where one is interested in a cluster-analysis of $S \cap Q$.

Clustering is a fundamental task in data analysis. It involves partitioning a given data set into subsets called *clusters*, such that similar elements end up in the same cluster. Often the data elements can be viewed as points in a geometric space, and similarity is measured by considering the distance between the points. We focus on clustering problems of the following type. Let S be a set of n points in $\mathbb{R}^d$, and let $k \geqslant 2$ be a natural number. A *k-clustering* of S is a partitioning $\mathcal{C}$ of S into at most k clusters. Let $\Phi(\mathcal{C})$ denote the *cost* of $\mathcal{C}$. The goal is now to find a clustering $\mathcal{C}$ that minimizes $\Phi(\mathcal{C})$. Many well-known geometric clustering problems are of this type. Among them

* MA is partly supported by Mikkel Thorup's Advanced Grant from the Danish Council for Independent Research under the Sapere Aude research career programme. MdB, KB, MM, and AM are supported by NWO grants 024.002.003, 612.001.207, 022.005.025, and 612.001.118 respectively.

is the k-center problem. In the *Euclidean k-center problem* $\Phi(\mathcal{C})$ is the maximum cost of any of the clusters $C \in \mathcal{C}$, where the cost of C is the radius of its smallest enclosing ball. Hence, in the Euclidean k-center problem we want to cover the point set S by k congruent balls of minimum radius. The *rectilinear k-center problem* is defined similarly except that one considers the L_∞-metric; thus we want to cover S by k congruent axis-aligned cubes[3] of minimum size. The k-center problem, including the important special case of the 2-center problem, has been studied extensively, both for the Euclidean case (e.g. [2,8,12,17,16,22]) and for the rectilinear case (e.g. [7,23]).

All papers mentioned above—in fact, all papers on clustering that we know of—consider clustering in the single-shot version. We are the first to study *range-clustering queries* on a point set S: given a query range Q and a parameter k, solve the given k-clustering problem on $S \cap Q$. We study this problem for the case where the query range is an axis-aligned box.

Background. Range-analysis queries can be seen as a very general form of range-aggregate queries. In a range-aggregate query, the goal is to compute some aggregate function $F(S \cap Q)$ over the points in the query range. The current state of the art typically deals with simple aggregate functions of the following form: each point $p \in S$ has a *weight* $w(p) \in \mathbb{R}$, and $F(S \cap Q) := \bigoplus_{p \in S \cap Q} w(p)$, where $\oplus$ is a semi-group operation. Such aggregate functions are *decomposable*, meaning that $F(A \cap B)$ can be computed from $F(A)$ and $F(B)$, which makes them easy to handle using existing data structures such as range trees.

Only some, mostly recent, papers describe data structures supporting non-decomposable analysis tasks. Several deal with finding the closest pair inside a query range (e.g. [1,10,13]). However, the closest pair does not give information about the global shape or distribution of $S \cap Q$, which is what our queries are about. The recent works by Brass *et al.* [5] and by Arya *et al.* [4] are more related to our paper. Brass *et al.* [5] present data structures for finding extent measures, such the width, area or perimeter of the convex hull of $S \cap Q$, or the smallest enclosing disk. (Khare *et al.* [18] improve the result on smallest-enclosing-disk queries.) These measures are strictly speaking not decomposable, but they depend only on the convex hull of $S \cap Q$ and convex hulls are decomposable. A related result is by Nekrich and Smid [20], who present a data structure that returns an ε-coreset inside a query range. The measure studied by Arya *et al.* [4], namely the length of the minimum spanning tree of $S \cap Q$, cannot be computed form the convex hull either: like our range-clustering queries, it requires more information about the structure of the point set. Thus our paper continues the direction set out by Arya *et al.*, which is to design data structures for more complicated analysis tasks on $S \cap Q$.

Contributions. Our main result is a general method to answer *approximate* orthogonal range-clustering queries in $\mathbb{R}^d$. Here the query specifies (besides the query box Q and the number of clusters k) a value $\varepsilon > 0$; the goal then is to compute a k-clustering $\mathcal{C}$ of $S \cap Q$ with $\Phi(\mathcal{C}) \leqslant (1+\varepsilon) \cdot \Phi(\mathcal{C}_{\text{opt}})$, where $\mathcal{C}_{\text{opt}}$ is an optimal clustering for $S \cap Q$. Our method works by computing a sample $R \subseteq S \cap Q$ such that solving the problem on R gives us the desired approximate solution. We show that for a large class of cost functions Φ we can find such a sample of size only $O(k(f(k)/\varepsilon)^d)$, where $f(k)$ is a function that only depends on the number of clusters. This is similar to the approach taken by Har-Peled and Mazumdar [15], who solve the (single-shot) approximate k-means and k-median problem efficiently by generating a coreset of size $O((k/\varepsilon^d) \cdot \log n)$. A key step in our method is

[3] Throughout the paper, when we speak of cubes (or squares, or rectangles, or boxes) we always mean axis-aligned cubes (or squares, or rectangles, or boxes).

a procedure to efficiently compute a lower bound on the value of an optimal solution within the query range. The class of clustering problems to which our method applies includes the k-center problem in any L_p-metric, variants of the k-center problem where we want to minimize the sum (rather than maximum) of the cluster radii, and the 2-dimensional problem where we want to minimize the maximum or sum of the perimeters of the clusters. Our technique allows us, for instance, to answer rectilinear k-center queries in the plane in $O((1/\varepsilon)\log n + 1/\varepsilon^2)$ for $k = 2$ or 3, in $O((1/\varepsilon)\log n + (1/\varepsilon^2)\text{polylog}(1/\varepsilon))$ for $k = 4$ or 5, and in $O((k/\varepsilon)\log n + (k/\varepsilon)^{O(\sqrt{k})})$ time for $k > 3$. We also show that for the rectilinear (or Euclidean) k-center problem, our method can be extended to deal with the capacitated version of the problem. In the capacitated version each cluster should not contain more than $\alpha \cdot (|S \cap Q|/k)$ points, for a given $\alpha > 1$.

In the second part of the paper we turn our attention to exact solutions to range-clustering queries. Here we focus on rectilinear *k-center queries*—that is, range-clustering queries for the rectilinear k-center problem—in $\mathbb{R}^1$ and $\mathbb{R}^2$. We present two linear-size data structures for queries in $\mathbb{R}^1$; one has $O(k^2 \log^2 n)$ query time, the other has $O(3^k \log n)$ query time. For queries in $\mathbb{R}^2$ we present a data structure that answers 2-center queries in $O(\log n)$ time, and one that answers 3-center queries in $O(\log^2 n)$ time. Both data structures use $O(n \log^\varepsilon n)$ storage, where $\varepsilon > 0$ is an arbitrary small (but fixed) constant.

2 Approximate Range-Clustering Queries

In this section we present a general method to answer approximate range-clustering queries. We start by defining the class of clustering problems to which it applies.

Let S be a set of n points in $\mathbb{R}^d$ and let $\text{Part}(S)$ be the set of all partitions of S. Let $\text{Part}_k(S)$ be the set of all partitions into at most k subsets, that is, all k-clusterings of S. Let $\Phi : \text{Part}(S) \mapsto \mathbb{R}_{\geqslant 0}$ be the cost function defining our clustering problem, and define

$$\textsc{Opt}_k(S) := \min_{\mathcal{C} \in \text{Part}_k(S)} \Phi(\mathcal{C})$$

to be the minimum cost of any k-clustering. Thus the goal of a range-clustering query with query range Q and parameter $k \geqslant 2$ is to compute a clustering $\mathcal{C} \in \text{Part}_k(S_Q)$ such that $\Phi(\mathcal{C}) = \textsc{Opt}_k(S_Q)$, where $S_Q := S \cap Q$. The method presented in this section gives an approximate answer to such a query: for a given constant $\varepsilon > 0$, which can be specified as part of the query, the method will report a clustering $\mathcal{C} \in \text{Part}_k(S_Q)$ with $\Phi(\mathcal{C}) \leqslant (1+\varepsilon) \cdot \textsc{Opt}_k(S_Q)$.

To define the class of clusterings to which our method applies, we will need the concept of r-packings [14]. Actually, we will use a slightly weaker variant, which we define as follows. Let $|pq|$ denote the Euclidean distance between two points p and q. A subset $R \subseteq P$ of a point set P is called a *weak r-packing* for P, for some $r > 0$, if for any point $p \in P$ there exists a *packing point* $q \in R$ such that $|pq| \leqslant r$. (The difference with standard r-packing is that we do not require that $|qq'| > r$ for any two points $q, q' \in R$.) The clustering problems to which our method applies are the ones whose cost function is *regular*, as defined next.

Definition 1. *A cost function $\Phi : \text{Part}(S) \mapsto \mathbb{R}_{\geqslant 0}$ is called* $(c, f(k))$-regular*, if there is constant c and function $f : \mathbb{N}_{\geqslant 2} \mapsto \mathbb{R}_{\geqslant 0}$ such that the following holds.*

– *For any clustering $\mathcal{C} \in \text{Part}(S)$, we have*

$$\Phi(\mathcal{C}) \geqslant c \cdot \max_{C \in \mathcal{C}} \text{diam}(C),$$

where $\mathrm{diam}(C) = \max_{p,q \in C} |pq|$ *denotes the Euclidean diameter of the cluster* C*. We call this the* diameter-sensitivity *property.*

- *For any subset* $S' \subseteq S$*, any weak* r*-packing* R *of* S'*, and any* $k \geqslant 2$*, we have that*

$$\textsc{Opt}_k(R) \leqslant \textsc{Opt}_k(S') \leqslant \textsc{Opt}_k(R) + r \cdot f(k).$$

Moreover, given a k*-clustering* $\mathcal{C} \in \mathrm{Part}_k(R)$ *we can compute a* k*-clustering* $\mathcal{C}^* \in \mathrm{Part}_k(S')$ *with* $\Phi(\mathcal{C}^*) \leqslant \Phi(\mathcal{C}) + r \cdot f(k)$ *in time* $T_{expand}(n,k)$*. We call this the* expansion *property.*

Examples. Many clustering problems have regular cost functions, in particular when the cost function is the aggregation—the sum, for instance, or the maximum—of the costs of the individual clusters. Next we give some examples.

The k*-center problem in any* L_p*-metric.* For a cluster C, let $\mathrm{radius}_p(C)$ denote the radius of the minimum enclosing ball of C in the L_p-metric. In the L_∞, for instance, $\mathrm{radius}_p(C)$ is half the edge length of a minimum enclosing axis-aligned cube of C. Then the cost of a clustering $\mathcal{C}$ for the k-center problem in the L_p-metric is $\Phi_p^{\max}(\mathcal{C}) = \max_{C \in \mathcal{C}} \mathrm{radius}_p(C)$. One easily verifies that the cost function for the rectilinear k-center problem is $(1/(2\sqrt{d}), 1)$-regular, and for the Euclidean k-center problem it is $(1/2, 1)$-regular. Moreover, $T_{\mathrm{expand}}(n,k) = O(k)$ for the k-center problem, since we just have so scale each ball by adding r to its radius.[4] (In fact $\Phi_p^{\max}(\mathcal{C})$ is regular for any p.)

Min-sum variants of the k*-center problem.* In the k-center problem the goal is to minimize $\max_{C \in \mathcal{C}} \mathrm{radius}_p(C)$. Instead we can also minimize $\Phi_p^{\mathrm{sum}}(\mathcal{C}) := \sum_{C \in \mathcal{C}} \mathrm{radius}_p(C)$, the sum of the cluster radii. Also these costs functions are regular; the only difference is that the expansion property is now satisfied with $f(k) = k$, instead of with $f(k) = 1$. Another interesting variant is to minimize $\left(\sum_{C \in \mathcal{C}} \mathrm{radius}_2(C)^2\right)^{1/2}$, which is $(1/(2\sqrt{d}), \sqrt{k})$-regular.

Minimum perimeter k*-clustering problems.* For a cluster C of points in $\mathbb{R}^2$, define $\mathrm{per}(C)$ to be the length of the perimeter of the convex hull of C. In the minimum perimeter-sum clustering problem the goal is to compute a k-clustering $\mathcal{C}$ such that $\Phi_{\mathrm{per}} := \sum_{C \in \mathcal{C}} \mathrm{per}(C)$ is minimized [6]. This cost function is $(2, 2\pi k)$-regular. Indeed, if we expand the polygons in a clustering $\mathcal{C}$ of a weak r-packing R by taking the Minkowski sum with a disk of radius r, then the resulting shapes cover all the points in S. Each perimeter increases by $2\pi r$ in this process. To obtain a clustering, we then assign each point to the cluster of its closest packing point, so $T_{\mathrm{expand}}(n,k) = O(n \log n)$.

Non-regular costs functions. Even though many clustering problems have regular costs functions, not all clustering problems do. For example, the k-means problem does not have a regular cost function. Minimizing the the max or sum of the areas of the convex hulls of the clusters is not regular either.

Our data structure and query algorithm. We start with a high-level overview of our approach. Let S be the given point set on which we want to answer range-clustering queries, and let Q be the query range. From now on we use S_Q as a shorthand for $S \cap Q$. We assume we have an

[4] This time bound only accounts for reporting the set of cubes that define the clustering. If we want to report the clusters explicitly, we need to add an $O(n)$ term.

algorithm $\textsc{SingleShotClustering}(P, k)$ that computes an optimal solution to the k-clustering problem (for the given cost function Φ) on a given point set P. (Actually, it is good enough if $\textsc{SingleShotClustering}(P, k)$ gives a $(1+\varepsilon)$-approximation.) Our query algorithm proceeds as follows.

Algorithm 1 $\textsc{ClusterQuery}(k, Q, \varepsilon)$.

1. Compute a lower bound $\textsc{lb}$ on $\textsc{Opt}_k(S_Q)$.
2. Set $r := \varepsilon \cdot \textsc{lb}/f(k)$ and compute a weak r-packing R on S_Q.
3. $\mathcal{C} := \textsc{SingleShotClustering}(R, k)$.
4. Expand $\mathcal{C}$ into a k-clustering $\mathcal{C}^*$ of cost at most $\Phi(\mathcal{C}) + r \cdot f(k)$ for S_Q.
5. Return $\mathcal{C}^*$.

Note that Step 4 is possible because Φ is $(c, f(k))$-regular. The following lemma is immediate.

Lemma 1. $\Phi(\mathcal{C}^*) \leqslant (1+\varepsilon) \cdot \textsc{Opt}_k(S_Q)$.

Next we show how to perform Step 1 and 2: we will describe a data structure that allows us to compute a suitable lower bound $\textsc{lb}$ and a corresponding weak r-packing, such that the size of the r-packing depends only on ε and k but not on $|S_Q|$.

Our lower bound and r-packing computations are based on so-called cube covers. A *cube cover* of S_Q is a collection $\mathcal{B}$ of interior-disjoint cubes that together cover all the points in S_Q and such that each $B \in \mathcal{B}$ contains at least one point from S_Q (in its interior or on its boundary). Define the size of a cube B, denoted by $\text{size}(B)$, to be its edge length. The following lemma follows immediately from the fact that the diameter of a cube B in $\mathbb{R}^d$ is $\sqrt{d} \cdot \text{size}(B)$.

Lemma 2. *Let $\mathcal{B}$ be a cube cover of S_Q such that $\text{size}(B) \leqslant r/\sqrt{d}$ for all $B \in \mathcal{B}$. Then any subset $R \subseteq S_Q$ containing a point from each cube $B \in \mathcal{B}$ is a weak r-packing for S.*

Our next lemma shows we can find a lower bound on $\textsc{Opt}_k(S_Q)$ from a suitable cube cover.

Lemma 3. *Suppose the cost function Φ is $(c, f(k))$-regular. Let $\mathcal{B}$ be a cube cover of S_Q such that $|\mathcal{B}| > k2^d$. Then $\textsc{Opt}_k(S_Q) \geqslant c \cdot \min_{B \in \mathcal{B}} \text{size}(B)$.*

Proof. For two cubes B and B' such that the maximum x_i-coordinate of B is at most the minimum x_i-coordinate of B', we say that B is *i-below* B' and B' is *i-above* B. We denote this relation by $B \prec_i B'$. Now consider an optimal k-clustering $\mathcal{C}_{\text{opt}}$ of S_Q. By the pigeonhole principle, there is a cluster $C \in \mathcal{C}_{\text{opt}}$ containing points from at least $2^d + 1$ cubes. Let $\mathcal{B}_C$ be the set of cubes that contain at least one point in C.

Clearly, if there are cubes $B, B', B'' \in \mathcal{B}_C$ such that $B' \prec_i B \prec_i B''$ for some $1 \leqslant i \leqslant d$, then the cluster C contains two points (namely from B' and B'') at distance at least $\text{size}(B)$ from each other. Since Φ is $(c, f(k))$-regular this implies that $\Phi(\mathcal{C}_{\text{opt}}) \geqslant c \cdot \text{size}(B)$, which proves the lemma.

Now suppose for a contradiction that such a triple B', B, B'' does not exist. Then we can define a characteristic vector $\Gamma(B) = (\Gamma_1(B), \ldots, \Gamma_d(B))$ for each cube $B \in \mathcal{B}_C$, as follows:

$$\Gamma_i(B) = \begin{cases} 0 \text{ if no cube } B' \in \mathcal{B}_C \text{ is } i\text{-below } B \\ 1 \text{ otherwise} \end{cases}$$

Since the number of distinct characteristic vectors is $2^d < |B_C|$, there must be two cubes $B_1, B_2 \in B_C$ with identical characteristic vectors. However, any two interior-disjoint cubes can be separated by an axis-aligned hyperplane, so there is at least one $i \in \{1, \dots, d\}$ such that $B_1 \prec_i B_2$ or $B_2 \prec_i B_1$. Assume without loss of generality that $B_1 \prec_i B_2$, so $\Gamma_i(B_2) = 1$. Since $\Gamma(B_1) = \Gamma(B_2)$ there must be a cube B_3 with $B_3 \prec_i B_1$. But then we have a triple $B_3 \prec_i B_1 \prec_i B_2$, which is a contradiction.

Next we show how to efficiently perform Steps 1 and 2 of ClusterQuery. Our algorithm uses a compressed octree $\mathcal{T}(S)$ on the point set S, which we briefly describe next.

For an integer s, let G_s denote the grid in $\mathbb{R}^d$ whose cells have size 2^s and for which the origin O is a grid point. A *canonical cube* is any cube that is a cell of a grid G_s, for some integer s. A *compressed octree* on a point set S in $\mathbb{R}^d$ contained in a canonical cube B is a tree-like structure defined recursively, as follows.

- If $|S| \leqslant 1$, then $\mathcal{T}(S)$ consists of a single leaf node, which corresponds to the cube B.
- If $|S| > 1$, then consider the cubes $B_1, \dots, B_{2^d}$ that result from cutting B into 2^d equal-sized cubes.
  - If at least two of the cubes U_i contain at least one point from S then $\mathcal{T}(S)$ consists of a root node with 2^d children $v_1, \dots, v_{2^d}$, where v_i is the root of a compressed octree for[5] $B_i \cap S$.
  - If all points from S lie in the same cube B_i, then let $B_{\text{in}} \subseteq B_i$ be the smallest canonical cube containing all points in S. Now $\mathcal{T}(S)$ consists of a root node with two children: one child v which is the root of a compressed octree for S inside B_{in}, and one leaf node w which represents the donut region $B \setminus B_{\text{in}}$.

A compressed octree for a set S of n points can be computed in $O(n \log n)$ time, assuming a model of computation where the smallest canonical cube of two points can be computed in $O(1)$ time [14, Theorem 2.23]. For a node $v \in \mathcal{T}(S)$, we denote the cube or donut corresponding to v by B_v, and we define $S_v := B_v \cap S$. It will be convenient to slightly modify the compressed quadtree by removing all nodes v such that $S_v = \emptyset$. (These nodes must be leaves.) Note that this removes all nodes v such that B_v is a donut. As a result, the parent of such a donut node now has only one child, w; we remove w and link the parent of w directly to w's (non-empty) children. The modified tree $\mathcal{T}(S)$—with a slight abuse of terminology we still refer to $\mathcal{T}(S)$ as a compressed octree—has the property that any internal node has at least two children. We augment $\mathcal{T}(S)$ by storing at each node v an arbitrary point $p \in B_v \cap S$.

Our algorithm descends into $\mathcal{T}(S)$ to find a cube cover $\mathcal{B}$ of S_Q consisting of canonical cubes, such that $\mathcal{B}$ gives us a lower bound on $\text{Opt}_k(S_Q)$. In a second phase, the algorithm then refines the cubes in the cover until they are small enough so that, if we select one point from each cube, we get a weak r-packing of S_Q for the appropriate value of r. The details are described in Algorithm 2, where we assume for simplicity that $|S_Q| > 1$. (The case $|S_Q| \leqslant 1$ is easy to check and handle. In addition, the algorithm will need several supporting data structures, which we will describe them later.)

Note that we continue the loop in lines 3–3 until we collect $k2^{2d}$ cubes (and not $k2^d$, as Lemma 3 would suggest) and that in line 5 we take the maximum cube size (instead of the minimum, as Lemma 3 would suggest).

[5] Here we assume that points on the boundary between cubes are assigned to one of these cubes in a consistent manner.

Algorithm 2 Algorithm for steps 1 and 2 of ClusterQuery, for a $(c, f(k))$-regular cost function.

1. $\mathcal{B}_{\text{inner}} := B_{\text{root}(\mathcal{T}(S))}$ and $\mathcal{B}_{\text{leaf}} := \emptyset$.
2. $\triangleright$ Phase 1: Compute a lower bound on $\text{Opt}_k(S_Q)$.
3. **While** $|\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}| \leqslant k2^{2d}$ and $\mathcal{B}_{\text{inner}} \neq \emptyset$ **do**
   - (i) Remove a largest cube B_v from $\mathcal{B}_{\text{inner}}$. Let v be the corresponding node.
   - (ii) **If** $B_v \not\subseteq Q$ **then**
     - (i) Compute $\text{bb}(S_Q \cap B_v)$, the bounding box of $S_Q \cap B_v$.
     - (ii) Find the deepest node u such that $\text{bb}(S_Q \cap B_v) \subseteq B_u$ and set $v := u$.
   - (iii) **EndIf**
   - (iv) For each child w of v such that $B_w \cap S_Q \neq \emptyset$, insert B_w into $\mathcal{B}_{\text{inner}}$ if w is an internal node and insert B_w into $\mathcal{B}_{\text{leaf}}$ if w is a leaf node.
4. **EndWhile**
5. $\text{LB} := c \cdot \max_{B_v \in \mathcal{B}_{\text{inner}}} \text{size}(B_v)$.
6. $\triangleright$ Phase 2: Compute a suitable weak r-packing.
7. $r := \varepsilon \cdot \text{LB}/f(k)$.
8. **While** $\mathcal{B}_{\text{inner}} \neq \emptyset$ **do**
   - (i) Remove a cube B_v from $\mathcal{B}_{\text{inner}}$ and handle it as in lines 3–3, with the following change: if $\text{size}(B_w) \leqslant r/\sqrt{d}$ then always insert B_w into $\mathcal{B}_{\text{leaf}}$ (not into $\mathcal{B}_{\text{inner}}$).
9. **EndWhile**
10. For each cube $B_v \in \mathcal{B}_{\text{leaf}}$ pick a point in $S_Q \cap B_v$ and put it into R_Q.
11. Return R_Q.

Lemma 4. *The value* LB *computed by Algorithm 2 is a correct lower bound on* $\text{Opt}_k(S_Q)$*. In addition, the set* R_Q *is a weak* r*-packing for* $r = \varepsilon \cdot \text{LB}/f(k)$ *of size* $O(k(f(k)/(c\,\varepsilon))^d)$*.*

Proof. As the first step to prove that LB is a correct lower bound, we claim that the loop in lines 3–3 maintains the following invariant: (i) $\bigcup(\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}})$ contains all points in S_Q, and (ii) each $B \in \mathcal{B}_{\text{inner}}$ contains at least two points from S_Q and each $B \in \mathcal{B}_{\text{leaf}}$ contains exactly one point from S_Q. This is trivially true before the loop starts, under our assumption that $|S_Q| \geqslant 2$. Now suppose we handle a cube $B_v \in \mathcal{B}_{\text{inner}}$. If $B_v \subseteq Q$ then we insert the cubes B_w of all children into $\mathcal{B}_{\text{inner}}$ or $\mathcal{B}_{\text{leaf}}$, which restores the invariant. If $B_v \not\subseteq Q$ then we first replace v by u. The condition $\text{bb}(S_Q \cap B_v) \subseteq B_u$ guarantees that all points of S_Q in B_v are also in B_u. Hence, if we then insert the cubes B_w of u's children into $\mathcal{B}_{\text{inner}}$ or $\mathcal{B}_{\text{leaf}}$, we restore the invariant. Thus at any time, and in particular after the loop, the set $\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$ is a cube cover of S_Q.

To complete the proof that LB is a correct lower bound we do not work with the set $\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$ directly, but we work with a set $\mathcal{B}$ defined as follows. For a cube $B_v \in \mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$, define $\text{parent}(B_v)$ to be the cube B_u corresponding to the parent node u of v. For each cube $B_v \in \mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$ we put one cube into $\mathcal{B}$, as follows. If there is another cube $B_w \in \mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$ such that $\text{parent}(B_w) \subsetneq \text{parent}(B_v)$, then we put B_v itself into $\mathcal{B}$, and otherwise we put $\text{parent}(B_v)$ into $\mathcal{B}$. Finally, we remove all duplicates from $\mathcal{B}$. Since $\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$ is a cube cover for S_Q—that is, the cubes in $\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$ are disjoint and they cover all points in S_Q—the same is true for $\mathcal{B}$. Moreover, the only duplicates in $\mathcal{B}$ are cubes that are the parent of multiple nodes in $\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$, and so $|\mathcal{B}| \geqslant |\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}|/2^d > k2^d$. By Lemma 3 we have $\text{Opt}_k(S_Q) \geqslant c \cdot \min_{B_v \in \mathcal{B}} \text{size}(B_v)$.

It remains to argue that $\min_{B_v \in \mathcal{B}} \text{size}(B_v) \geqslant \max_{B_v \in \mathcal{B}_{\text{inner}}} \text{size}(B_v)$. We prove this by contradiction. Hence, we assume $\min_{B_v \in \mathcal{B}} \text{size}(B_v) < \max_{B_v \in \mathcal{B}_{\text{inner}}} \text{size}(B_v)$ and we define $B :=$

$\arg\min_{B_v \in \mathcal{B}} \mathrm{size}(B_v)$ and $B' := \arg\max_{B_v \in \mathcal{B}_{\mathrm{inner}}} \mathrm{size}(B_v)$. Note that for any cube $B_v \in \mathcal{B}$ either B_v itself is in $\mathcal{B}_{\mathrm{inner}} \cup \mathcal{B}_{\mathrm{leaf}}$ or $B_v = \mathrm{parent}(B_w)$ for some cube $B_w \in \mathcal{B}_{\mathrm{inner}} \cup \mathcal{B}_{\mathrm{leaf}}$. We now make the following case distinction.

Case I: $B = \mathrm{parent}(B_w)$ for some cube $B_w \in \mathcal{B}_{\mathrm{inner}} \cup \mathcal{B}_{\mathrm{leaf}}$. But this is an immediate contradiction since Algorithm 2 would have to split B' before splitting B.

Case II: $B \in \mathcal{B}_{\mathrm{inner}} \cup \mathcal{B}_{\mathrm{leaf}}$. Because B itself was put into $\mathcal{B}$ and not $\mathrm{parent}(B)$, there exists a cube $B_w \in \mathcal{B}_{\mathrm{inner}} \cup \mathcal{B}_{\mathrm{leaf}}$ such that $\mathrm{parent}(B) \supsetneq \mathrm{parent}(B_w)$, which means $\mathrm{size}(\mathrm{parent}(B_w)) < \mathrm{size}(\mathrm{parent}(B))$. In order to complete the proof, it suffices to show that $\mathrm{size}(\mathrm{parent}(B_w)) \leqslant \mathrm{size}(B)$. Indeed, since B' has not been split by Algorithm 2 (because $B' \in \mathcal{B}_{\mathrm{inner}}$) we know that $\mathrm{size}(B') \leqslant \mathrm{size}(\mathrm{parent}(B_w))$. This inequality along with the inequality $\mathrm{size}(\mathrm{parent}(B_w)) \leqslant \mathrm{size}(B)$ imply that $\mathrm{size}(B') \leqslant \mathrm{size}(B)$ which is in contradiction with $\mathrm{size}(B) < \mathrm{size}(B')$. To show that $\mathrm{size}(\mathrm{parent}(B_w)) \leqslant \mathrm{size}(B)$ we consider the following two subcases. (i) $\mathrm{parent}(B)$ is a degree-1 node. This means that $\mathrm{parent}(B)$ corresponds to a cube that was split into a donut and the cube corresponding to B. Since the cube corresponding to B_w must be completely inside the cube corresponding to $\mathrm{parent}(B)$ (because $\mathrm{size}(\mathrm{parent}(B_w)) < \mathrm{size}(\mathrm{parent}(B))$) and a donut is empty we conclude that the cube corresponding to B_w must be completely inside the cube corresponding to B. Hence, $\mathrm{size}(\mathrm{parent}(B_w)) \leqslant \mathrm{size}(B)$. (ii) $\mathrm{parent}(B)$ is not a degree-1 node. The inequality $\mathrm{size}(\mathrm{parent}(B_w)) < \mathrm{size}(\mathrm{parent}(B))$ along with the fact that $\mathrm{parent}(B)$ is not a degree-1 node imply that $\mathrm{size}(\mathrm{parent}(B_w)) \leqslant \mathrm{size}(B)$.

This completes the proof that LB is a correct lower bound. Next we prove that R_Q is a weak r-packing for $r = \varepsilon \cdot \mathrm{LB}/f(k)$. Observe that after the loop in lines 8–9, the set $\mathcal{B}_{\mathrm{leaf}}$ is still a cube cover of S_Q. Moreover, each cube $B_v \in \mathcal{B}_{\mathrm{leaf}}$ either contains a single point from S_Q or its size is at most $r/\sqrt{d}$. Lemma 2 then implies that R_Q is a weak r-packing for the desired value of r.

It remains to bound the size of R_Q. To this end we note that at each iteration of the loop in lines 3–3 the size of $\mathcal{B}_{\mathrm{inner}} \cup \mathcal{B}_{\mathrm{leaf}}$ increases by at most $2^d - 1$, so after the loop we have $|\mathcal{B}_{\mathrm{inner}} \cup \mathcal{B}_{\mathrm{leaf}}| \leqslant k2^{2d} + 2^d - 1$. The loop in lines 8–9 replaces each cube $B_v \in \mathcal{B}_{\mathrm{inner}}$ by a number of smaller cubes. Since $\mathrm{LB} = c \cdot \max_{B_v \in \mathcal{B}_{\mathrm{inner}}} \mathrm{size}(B_v)$ and $r = \varepsilon \cdot \mathrm{LB}/f(k)$, each cube B_v is replaced by only $O((f(k)2^d\sqrt{d}/(c\,\varepsilon))^d)$ smaller cubes. Since d is a fixed constant, the total number of cubes we end up with (which is the same as the size of the r-packing) is $O(k(f(k)/(c\,\varepsilon))^d)$.

Lemma 4, together with Lemma 1, establishes the correctness of our approach. To achieve a good running time, we need a few supporting data structures.

– We need a data structure that can answer the following queries: given a query box Z, find the deepest node u in $\mathcal{T}(S)$ such that $Z \subseteq B_u$. With a *centroid-decomposition tree* $\mathcal{T}_{\mathrm{cd}}$ we can answer such queries in $O(\log n)$ time. A centroid-decomposition tree $\mathcal{T}_{\mathrm{cd}}$ on the compressed octree $\mathcal{T}(S)$ is defined as follows. View $\mathcal{T}(S)$ as an acyclic graph of maximum degree $2^d + 1$. Let v^* be a centroid of $\mathcal{T}(S)$, that is, v^* is a node whose removal splits $\mathcal{T}(S)$ into at most $2^d + 1$ subgraphs each containing at most half the nodes. We recursively construct centroid-decomposition trees $\mathcal{T}^1_{\mathrm{cd}}, \mathcal{T}^2_{\mathrm{cd}}, \ldots$ for each of the subgraphs. The centroid-decomposition tree $\mathcal{T}_{\mathrm{cd}}$ now consists of a root node corresponding to v^* that has $\mathcal{T}^1_{\mathrm{cd}}, \mathcal{T}^2_{\mathrm{cd}}, \ldots$ as subtrees. Note that one of the subtrees corresponds to the region outside B_{v^*}, while the other subtrees correspond to regions inside B_{v^*} (namely the cubes of the children of v^* in $\mathcal{T}(S)$).
With $\mathcal{T}_{\mathrm{cd}}$ we can answer the following queries in $O(\log n)$ time: given a query box Z, find the deepest node u in $\mathcal{T}(S)$ such that $Z \subseteq B_u$. We briefly sketch the (standard) procedure for this. First, check if $Z \subseteq B_{v^*}$, where v^* is the node of $\mathcal{T}(S)$ corresponding to the root of $\mathcal{T}_{\mathrm{cd}}$. If not,

recursively search the subtree $\mathcal{T}_{\text{cd}}^{j}$ corresponding to the region outside B_{v^*}. If $Z \subseteq B_{v^*}$ then check if v^* has a child w such that $Z \subseteq B_w$; if so, recurse on the corresponding subtree $\mathcal{T}_{\text{cd}}^{j'}$, and otherwise report B_{v^*} as the answer.

– We need a data structure $\mathcal{D}$ that can answer the following queries on S: given a query box Z and an integer $1 \leqslant i \leqslant d$, report a point in $S \cap Z$ with maximum x_i-coordinate, and one with minimum x_i-coordinate. It is possible to answer such queries in $O(\log^{d-1} n)$ time with a range tree (with fractional cascading), which uses $O(n \log^{d-1} n)$ storage. Note that this also allows us to compute the bounding box of $S \cap Z$ in $O(\log^{d-1} n)$ time. (In fact slightly better bounds are possible [19], but for simplicity we stick to using standard data structures.)

Lemma 5. *Algorithm 2 runs in* $O(k\,(f(k)/(c\,\varepsilon))^d + k\,((f(k)/(c\,\varepsilon))\log n)^{d-1})$ *time.*

Proof. We store the set $\mathcal{B}_{\text{inner}}$ in a priority queue base on the size of the cubes, so we can remove the cube of maximum size in $O(\log n)$ time. To handle a cube B_v in an iteration of the first while loop we need $O(\log^{d-1} n)$ time, which is the time needed to compute $\text{bb}(S_Q \cap B_v)$ using our supporting data structure $\mathcal{D}$. Next observe that each iteration of the loop increases the size of $\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$. When $B_v \subseteq Q$ this is clear, since every internal node in $\mathcal{T}(S)$ has at least two children. When $B_v \not\subseteq Q$ we first replace v by the deepest node u such that $\text{bb}(S_Q \cap B_v) \subseteq B_u$. This ensures that at least two of the children of u must contain a point in S_Q, so the size of $\mathcal{B}_{\text{inner}} \cup \mathcal{B}_{\text{leaf}}$ also increases in this case. We conclude that the number of iterations is bounded by $k2^{2d}$, and so the running time for Phase 1 is $O(k2^{2d} \log^{d-1} n)$.

To bound the time for Phase 2 we observe that the computation of $\text{bb}(S_Q \cap B_v)$ is only needed when $B_v \not\subseteq Q$. Similarly, we only need our supporting data structure $\mathcal{D}$ for picking a point from $S_Q \cap B_v$ in line 10 when $B_v \not\subseteq Q$. The total number of cubes that are handled and generated in Phase 2 is $O(k(f(k)/(c\ \varepsilon))^d)$, but the number of cubes that intersect the boundary of the query range Q is a factor $f(k)/\varepsilon$ smaller. Thus the total time for Phase 2 is $O(k\,(f(k)/(c\,\varepsilon))^d + k\,((f(k)/(c\,\varepsilon)))^{d-1} \log^{d-1} n)$.

This leads to the following theorem (where we use that T_{ss} is at least linear).

Theorem 1. *Let S be a set of n points in $\mathbb{R}^d$ and let Φ be a $(c, f(k))$-regular cost function. Suppose we have an algorithm that solves the given clustering problem on a set of m points in $T_{\text{ss}}(m)$ time. Then there is a data structure that uses $O(n \log^{d-1} n)$ storage such that, for a query range Q and query values $k \geqslant 2$ and $\varepsilon > 0$, we can compute a $(1+\varepsilon)$-approximate answer to a range-clustering query in time*

$$O\left(k\left(\frac{f(k)}{c\,\varepsilon}\cdot\log n\right)^{d-1} + T_{\text{ss}}\left(k\left(\frac{f(k)}{c\,\varepsilon}\right)^{d}, k\right) + T_{\text{expand}}(n,k)\right).$$

As an example application we consider k-center queries in the plane. (The result for rectilinear 2-center queries is actually inferior to the exact solution presented later.)

Corollary 1. *Let S be a set of n points in $\mathbb{R}^2$. There is a data structure that uses $O(n \log n)$ storage such that, for a query range Q and query values $k \geqslant 2$ and $\varepsilon > 0$, we can compute a $(1+\varepsilon)$-approximate answer to a k-center query within the following bounds:*

(i) For the rectilinear case with $k = 2$ or 3, the query time is $O((1/\varepsilon)\log n + 1/\varepsilon^2)$;

(ii) *For the rectilinear case with $k = 4$ or 5, the query time is*

$$O((1/\varepsilon) \log n + (1/\varepsilon^2) \cdot \text{polylog}(1/\varepsilon));$$

(iii) *For the Euclidean case with $k = 2$, the expected query time is*

$$O((1/\varepsilon) \log n + (1/\varepsilon^2) \log^2(1/\varepsilon));$$

(iv) *For the rectilinear case with $k > 5$ and the Euclidean case with $k > 2$ the query time is $O((k/\varepsilon) \log n + (k/\varepsilon)^{O(\sqrt{k})})$.*

Proof. Recall that the cost function for the k-center problem is $(1/(2\sqrt{d}), 1)$-regular for the rectilinear case and $(1/2, 1)$-regular for the Euclidean case. We now obtain our results by plugging in the appropriate algorithms for the single-shot version. For (i) we use the linear-time algorithm of Hoffmann [16], for (ii) we use the $O(n \cdot \text{polylog} n)$-time algorithms of Sharir and Welzl [23], for (iii) we use the $O(n \log^2 n)$-time randomized algorithm of Eppstein [12], and for (iv) we use the $n^{O(\sqrt{k})}$-time algorithm of Agarwal and Procopiuc [3].

3 Approximate Capacitated *k*-Center Queries

In this section we study the capacitated variant of the rectilinear k-center problem in the plane. In this variant we want to cover a set S of n points in $\mathbb{R}^2$ with k congruent squares of minimum size, under the condition that no square is assigned more than $\alpha \cdot n/k$ points, where $\alpha > 1$ is a given constant. For a capacitated rectilinear k-center query this means we want to assign no more than $\alpha \cdot |S_Q|/k$ points to each square. Our data structure will report a $(1+\varepsilon, 1+\delta)$-approximate answer to capacitated rectilinear k-center queries: given a query range Q, a natural number $k \geqslant 2$, a constant $\alpha > 1$, and real numbers $\varepsilon, \delta > 0$, it computes a set $\mathcal{C} = \{b_1, \dots, b_k\}$ of congruent squares such that:

- each b_i can be associated to a subset $C_i \subseteq S_Q \cap b_i$ such that $\{C_1, \dots, C_k\}$ is a k-clustering of S_Q and $|C_i| \leqslant (1+\delta)\alpha \cdot |S_Q|/k$; and
- the size of the squares in $\mathcal{C}$ is at most $(1+\varepsilon) \cdot \text{OPT}_k(S_Q, \alpha)$, where $\text{OPT}_k(S_Q, \alpha)$ is the value of an optimal solution to the problem on S_Q with capacity upper bound $U_Q := \alpha \cdot |S_Q|/k$.

Thus we allow ourselves to violate the capacity constraint by a factor $1 + \delta$.

To handle the capacity constraints, it is not sufficient to work with r-packings—we also need δ-approximations. Let P be a set of points in $\mathbb{R}^2$. A *δ-approximation* of P with respect to axis-aligned rectangles is a subset $A \subseteq P$ such that for any rectangle σ we have

$$\big|\, |P \cap \sigma|/|P| \; - \; |A \cap \sigma|/|A| \big| \leqslant \delta$$

From now on, whenever we speak of δ-approximations, we mean δ-approximations with respect to rectangles. Our method will use a special variant of the capacitated k-center problem, where we also have points that must be covered but do not count for the capacity:

Definition 2. *Let $R \cup A$ be a point set in $\mathbb{R}^2$, $k \geqslant 2$ a natural number, and U a capacity bound. The* 0/1-weighted capacitated k-center problem *in $\mathbb{R}^2$ is to compute a set $\mathcal{C} = \{b_1, \dots, b_k\}$ of congruent squares of minimum size where each b_i is associated to a subset $C_i \subseteq (R \cup A) \cap b_i$ such that $\{C_1, \dots, C_k\}$ is a k-clustering of $R \cup A$ and $|C_i \cap A| \leqslant U$.*

For a square b, let expand(b, r) denote the square b expanded by r on each side (so its radius in the L_∞-metric increases by r). Let 0/1-WeightedKCenter be an algorithm for the single-shot capacitated rectilinear k-center problem. Our query algorithm is as follows.

Algorithm 3 CapacitatedKCenterQuery$(k, Q, \alpha, \varepsilon, \delta)$.

1. Compute a lower bound LB on $\text{Opt}_k(S_Q)$.
2. Set $r := \varepsilon \cdot \text{LB}/f(k)$ and compute a weak r-packing R on S_Q.
3. Set $\delta_Q := \delta/16k^3$ and compute a δ_Q-approximation A_Q on S_Q.
4. Set $U := (1 + \delta/2) \cdot \alpha \cdot |A_Q|/k$ and $\mathcal{C} :=$ 0/1-WeightedKCenter$(R \cup A_Q, k, U)$.
5. $\mathcal{C}^* := \{\text{expand}(b, r) : b \in \mathcal{C}\}$.
6. Return $\mathcal{C}^*$.

Note that the lower bound computed in Step 1 is a lower bound on the uncapacitated problem (which is also a lower bound for the capacitated problem). Hence, for Step 1 and Step 2 we can use the algorithm from the previous section. How Step 3 is done will be explained later. First we show that the algorithm gives a $(1 + \varepsilon, 1 + \delta)$-approximate solution. We start by showing that we get a valid solution that violates the capacity constraint by at most a factor $1 + \delta$.

Lemma 6. *Let $\mathcal{C}^* := \{b_1, \ldots, b_k\}$ be the set of squares computed in Step 5. There exists a partition $\{C_1, \ldots, C_k\}$ of S_Q such that $C_i \subseteq b_i$ and $|C_i| \leqslant (1 + \delta) \cdot U_Q$ for each $1 \leqslant i \leqslant k$, and such a partition can be computed in $O(k^2 + n \log n)$ time.*

Proof. Since R is a weak r-packing, after expanding the squares in Step 5 they cover all points in S_Q. Next we show that we can assign the points in S_Q to the squares in $\mathcal{C}^*$ such that the capacities are not violated by more than a factor $1 + \delta$.

Since $\mathcal{C}$ is a solution to the 0/1-weighted capacitated problem on $R_Q \cup A_Q$, there is a partition $A_1, \ldots, A_k$ of A_Q such that $A_i \subset b_i$ and $|A_i| \leqslant U$ for all $1 \leqslant i \leqslant k$. Partition the plane into a collection $\mathcal{Z}$ of $O(k^2)$ cells by drawing the at most $2k$ vertical and $2k$ horizontal lines containing the edges of the squares in $\mathcal{C}$. Consider a cell $\sigma \in \mathcal{Z}$ and assume σ is inside j different squares $b_{i_1}, \ldots, b_{i_j} \in \mathcal{C}$. We can partition σ into j rectangular subcells $\sigma_1, \ldots, \sigma_j$ such that $|A_Q \cap \sigma_t| = |A_{i_t} \cap \sigma|$ for all $1 \leqslant t \leqslant j$: subcell σ_1 will contain the topmost $|A_{i_1} \cap \sigma|$ points from A_Q, subcell σ_2 will contain the next $|A_{i_2} \cap \sigma|$ points, and so on. The total time for this is $O(n_\sigma \log n_\sigma)$ time, where $n_\sigma := |A_Q \cap \sigma|$. We now assign all points from $S_Q \cap \sigma_{i_t}$ to the square b_{i_t}; in other words, we put the points from $S_Q \cap \sigma_{i_t}$ into the cluster C_{i_t}. If we do this for all regions $\sigma \in \mathcal{Z}$, we obtain the desired partition $\{C_1, \ldots, C_k\}$ of S_Q.

It remains to prove that $|C_i| \leqslant (1 + \delta) \cdot U_Q$ for each $1 \leqslant i \leqslant k$. Let $\mathcal{Z}_i$ be the set of all subcells assigned to C_i. Observe that $\sum_{\sigma \in \mathcal{Z}_i} |A_Q \cap \sigma| = |A_i| \leqslant U$ and that[6] $|\mathcal{Z}_i| \leqslant 8k^2$. Moreover, since A_Q is a δ_Q-approximation for S_Q we have

$$|S_Q \cap \sigma| \leqslant \delta_Q \cdot |S_Q| + |A_Q \cap \sigma| \cdot \frac{|S_Q|}{|A_Q|}.$$

[6] In fact, the description above would give $|\mathcal{Z}_i| \leqslant 4k^2$. However, in degenerate cases we may need two subcells for some A_{i_t} when we subdivide a cell σ, increasing the number of subcells in $\mathcal{Z}_i$ by at most a factor of 2.

Hence,

$$\begin{aligned}
|C_i| &= \textstyle\sum_{\sigma\in\mathcal{Z}_i} |S_Q \cap \sigma| \\
&\leqslant \textstyle\sum_{\sigma\in\mathcal{Z}_i} \left(\delta_Q \cdot |S_Q| + |A_Q \cap \sigma| \cdot \frac{|S_Q|}{|A_Q|}\right) \\
&\leqslant |\mathcal{Z}_i| \cdot \delta_Q \cdot |S_Q| + \textstyle\sum_{\sigma\in\mathcal{Z}_i} |A_Q \cap \sigma| \cdot \frac{|S_Q|}{|A_Q|} \\
&\leqslant 8k^2 \cdot \delta_Q \cdot |S_Q| + U \cdot \frac{|S_Q|}{|A_Q|} \\
&\leqslant (\delta/(2k)) \cdot |S_Q| + \big((1+\delta/2) \cdot \alpha \cdot |A_Q|/k\big) \cdot \frac{|S_Q|}{|A_Q|} \\
&\leqslant (\delta/2) \cdot |S_Q|/k + (1+\delta/2) \cdot \alpha \cdot |S_Q|/k \\
&\leqslant (1+\delta) \cdot U_Q \qquad \text{(since } \alpha \geqslant 1\text{)}
\end{aligned}$$

To finish the proof, it remains to observe that the assignment of points to the expanded squares described above can easily be done in $O(k^2 + n \log n)$ time.

We also need to prove that we get a $(1+\varepsilon)$-approximate solution. To this end, it suffices to show that an optimal solution $\mathcal{C}_{\text{opt}}$ to the problem on S_Q is a valid solution on $R \cup A_Q$. We can prove this by a similar approach as in the proof of the previous lemma.

Lemma 7. *The size of the squares in $\mathcal{C}^*$ is at most $(1+\varepsilon) \cdot \textsc{Opt}_k(S_Q, \alpha)$.*

Proof. We show that an optimal solution $\mathcal{C}_{\text{opt}}$ to the problem on S_Q is a valid solution on $R \cup A_Q$. Let $\{b_1, \ldots, b_k\}$ and $\{C_1, \ldots, C_k\}$ be the sets of squares and their corresponding clusters in a solution of value $\textsc{Opt}_k(S_Q, \alpha)$ for S_Q. We claim that we can assign the points in A_Q to the squares b_i such that no square is assigned more than U points, where $U := (1+\delta/2)\alpha \cdot |A_Q|/k$. We can do this following a similar approach as in the proof of Lemma 6: we partition the plane into $O(k^2)$ cells, which we partition further into subcells that are assigned to squares b_i such that $\sum_{\sigma\in\mathcal{Z}_i} |S_Q \cap \sigma| = |C_i|$, where $\mathcal{Z}_i$ is the collection of subcells assigned to b_i. Then for each b_i we have

$$\begin{aligned}
\textstyle\sum_{\sigma\in\mathcal{Z}_i} |A_Q \cap \sigma| &\leqslant \textstyle\sum_{\sigma\in\mathcal{Z}_i} \left(\delta_Q \cdot |A_Q| + |S_Q \cap \sigma| \cdot \frac{|A_Q|}{|S_Q|}\right) \\
&\leqslant |\mathcal{Z}_i| \cdot \delta_Q \cdot |A_Q| + \textstyle\sum_{\sigma\in\mathcal{Z}_i} |S_Q \cap \sigma| \cdot \frac{|A_Q|}{|S_Q|} \\
&\leqslant 8k^2 \cdot \delta_Q \cdot |A_Q| + U_Q \cdot \frac{|A_Q|}{|S_Q|} \\
&\leqslant (\delta/(2k)) \cdot |A_Q| + \alpha \cdot |A_Q|/k \\
&\leqslant (\delta/2) \cdot \alpha \cdot |A_Q|/k + \alpha \cdot |A_Q|/k \\
&= U
\end{aligned}$$

To make CapacitatedKCenterQuery run efficiently, we need some more supporting data structures. In particular, we need to quickly compute a δ_Q-approximation within our range Q. To this end, we use the following data structures.

– We compute a collection $A_1, \ldots, A_{\log n}$, where A_i is a $(1/2^i)$-approximation on S, using the algorithm of Phillips [21]. This algorithm computes, given a planar point set P of size n and a parameter δ, a δ-approximation of size $O((1/\delta)\log^4(1/\delta) \cdot \text{polylog}(\log(1/\delta)))$ in time $O((n/\delta^3) \cdot \text{polylog}(1/\delta))$. We store each A_i in a data structure for orthogonal range-reporting queries. If we use a range tree with fractional cascading, the data structure uses $O(|A_i| \log |A_i|)$ space and we can report all the points in $A_i \cap Q$ in time $O(\log n + |A_i \cap Q|)$.

– We store S in a data structure for orthogonal range-counting queries. There is such a data structure that needs $O(n)$ space and it can answer orthogonal range-counting queries in $O(\log n)$ time [9].

We can now compute a δ_Q-approximation for S_Q as follows.

Algorithm 4 DeltaApprox(Q, δ_Q).

1. Find the smallest value for i such that $\frac{1}{2^i} \leqslant \frac{\delta_Q}{4}\frac{|S_Q|}{|S|}$, and compute $A := Q \cap A_i$.
2. Compute a $(\delta_Q/2)$-approximation A_Q on A using the algorithm by Phillips [21].
3. Return A_Q.

Lemma 8. DeltaApprox(Q, δ_Q) *computes a δ_Q-approximation of size*

$$O((1/\delta_Q) \cdot \mathrm{polylog}(1/\delta_Q))$$

on S_Q in time $O\left(\log^4(n/\delta_Q) \cdot \mathrm{polylog}(\log n/\delta_Q)\right)$.

To prove Lemma 8 we need the following additional lemma.

Lemma 9. *If A is a δ^*-approximation for a point set S in $\mathbb{R}^2$ with*

$$\delta^* \leqslant (\delta/2) \cdot (|S_Q|/|S|),$$

then $A_Q := Q \cap A$ is a δ-approximation for $S_Q := S \cap Q$.

Proof. Consider any rectangular range $\sigma \subset Q$. Since A is a δ^*-approximation for S we have

$$\left|\frac{|S \cap \sigma|}{|S|} - \frac{|A \cap \sigma|}{|A|}\right| \leqslant \delta^*,$$

and so

$$\left|\frac{|S \cap \sigma|}{|A \cap \sigma|} - \frac{|S|}{|A|}\right| \leqslant \delta^* \frac{|S|}{|A \cap \sigma|}. \tag{1}$$

Similarly, by considering Q itself as a range we know that

$$\left|\frac{|S_Q|}{|S|} - \frac{|A_Q|}{|A|}\right| \leqslant \delta^*$$

and so

$$\left|\frac{|S_Q|}{|A_Q|} - \frac{|S|}{|A|}\right| \leqslant \delta^* \frac{|S|}{|A_Q|}. \tag{2}$$

Combining Inequalities (1) and (2) and replacing δ^* with its upper bound we get

$$\left|\frac{|S_Q|}{|A_Q|} - \frac{|S \cap \sigma|}{|A \cap \sigma|}\right| \leqslant \frac{\delta}{2} \cdot \frac{|S_Q|}{|S|} \cdot \left(\frac{|S|}{|A_Q|} + \frac{|S|}{|A \cap \sigma|}\right) = \frac{\delta}{2} \cdot |S_Q| \cdot \left(\frac{1}{|A_Q|} + \frac{1}{|A \cap \sigma|}\right).$$

Since $A \cap \sigma = A_Q \cap \sigma$ and $S \cap \sigma = S_Q \cap \sigma$, and $|A_Q \cap \sigma| \leqslant |A_Q|$, we can now derive

$$\begin{aligned}\left|\frac{|A_Q \cap \sigma|}{|A_Q|} - \frac{|S_Q \cap \sigma|}{|S_Q|}\right| &\leqslant \frac{|A_Q \cap \sigma|}{|S_Q|} \cdot \left|\frac{|S_Q|}{|A_Q|} - \frac{|S \cap \sigma|}{|A \cap \sigma|}\right| \\ &\leqslant \frac{\delta}{2} \cdot \frac{|A_Q \cap \sigma|}{|S_Q|} \cdot |S_Q| \cdot \left(\frac{1}{|A_Q|} + \frac{1}{|A_Q \cap \sigma|}\right) \\ &\leqslant \frac{\delta}{2}\left(\frac{|A_Q \cap \sigma|}{|A_Q|} + 1\right) \\ &\leqslant \delta\end{aligned}$$

which proves the lemma.

Now we can prove Lemma 8.

Proof. By Lemma 9, the set A computed in Step 1 of DeltaApprox is a $(\delta_Q/2)$-approximation for S_Q. Computing A requires a range query on A_i, which takes $O(\log n + |A|)$ time. The $(1/2^i)$-approximation A_i computed (during preprocessing) by Phillips's algorithm has size

$$|A_i| \ = \ O(2^i \cdot \log^4(2^i) \cdot \mathrm{polylog}(\log 2^i)) \ = \ O(2^i \cdot \log^4 n \cdot \mathrm{polylog}(\log n)).$$

As i is the smallest value with $1/2^i \leqslant (\delta_Q/4) \cdot (|S_Q|/|S|)$, we have $1/2^i > (\delta_Q/8) \cdot (|S_Q|/|S|)$. Hence,

$$|S_Q|/|S| < (8/\delta_Q) \cdot (1/2^i)$$

Since A_i is a $(1/2^i)$-approximation for S we have

$$|A_i \cap Q| \leqslant (1/2^i) \cdot |A_i| + (|S_Q|/|S|) \cdot |A_i|$$

and so

$$\begin{aligned}|A| &= |A_i \cap Q| \\ &\leqslant (1/2^i) \cdot |A_i| + (|S_Q|/|S|) \cdot |A_i| \\ &\leqslant (1/2^i) \cdot |A_i| + (8/\delta_Q) \cdot (1/2^i) \cdot |A_i| \\ &\leqslant (1/2^i) \cdot |A_i| \cdot (1 + 8/\delta_Q) \\ &= O(\log^4 n \cdot \mathrm{polylog}(\log n)) \cdot O(1/\delta_Q) \\ &= O((1/\delta_Q) \cdot \log^4 n \cdot \mathrm{polylog}(\log n))\end{aligned}$$

Since a δ'-approximation of a δ''-approximation of a set P is a $(\delta' + \delta'')$-approximation of P, we see that the set A_Q computed in Step 2 is a δ_Q-approximation, as required. The time needed for Step 2 is $O((|A|/\delta_Q^3) \cdot \mathrm{polylog}(1/\delta_Q))$, which is

$$O((1/\delta_Q)^4 \cdot \log^4 n \cdot \mathrm{polylog}(\log n/\delta_Q)).$$

The only thing left is now an algorithm 0/1-WeightedKCenter$(R \cup A_Q, k, U)$ that solves the 0/1-weighted version of the capacitated rectilinear k-center problem. Here we use the following straightforward approach. Let $m := |R \cup A_Q|$. First we observe that at least one square in an optimal solution has points on opposite edges. Hence, to find the optimal size we can do a binary search over $O(m^2)$ values, namely the horizontal and vertical distances between any pair of points. Moreover, given a target size s we can push al squares such that each has a point on its bottom edge and a point on its left edge. Hence, to test if there is a solution of a given target size s, we only have to test $O(m^{2k})$ sets of k squares. To test such a set $\mathcal{C} = \{b_1, \dots, b_k\}$ of squares, we need to check if the squares cover all points in $R \cup A_Q$ and if we can assign the points to squares such that the capacity constraint is met. For the latter we need to solve a flow problem, which can be done in $O(m^2 k)$ time. More precisely, given a set $\mathcal{C} = \{b_1, \dots, b_k\}$ of k squares, a set P of m points, and a capacity upper bound U, and we have to decide if we can assign each point in P to a square in $\mathcal{C}$ containing it such that no square in $\mathcal{C}$ is assigned more than U points. We can model this as a flow problem in a standard manner. For completeness we describe how this is done.

We construct a flow network with source s and sink t, and one vertex v_p for each point $p \in A_Q$ and one vertex u_i for each square b_i. We add the following edges.

1. For each v_p, we add one edge with capacity 1 from s to v_p.
2. For each u_i we add one edge with capacity $|U|$ from u_i to t.
3. For each pair (p, b_i) where $p \in A_Q \cap b_i$ add an edge with capacity 1 from v_p to u_i.

We solve the flow problem using the Ford-Fulkerson algorithm which works in $O(|E| \cdot |f|)$ time, where $|E|$ is the number of the edges and $|f|$ is maximum flow value. In our problem, $|E| = O(mk)$ and $|f| = |U| \leqslant m$, which results in an $O(m^2 k)$ time bound.

Thus each step in the binary search takes $O(m^{2k+2}k)$, leading to an overall time complexity for 0/1-WeightedKCenter$(R \cup A_Q, k, U)$ of $O(m^{2k+2}k \log m)$, where $m = |R \cup A_Q| = O\left(k/\varepsilon^2 + (1/\delta_Q) \cdot \text{polylog}(1/\delta_Q)\right)$, where $\delta_Q = \Theta(\delta/k^3)$.

The following theorem summarizes the results in this section.

Theorem 2. *Let S be a set of n points in $\mathbb{R}^2$. There is a data structure that uses $O(n \log n)$ storage such that, for a query range Q and query values $k \geqslant 2$, $\varepsilon > 0$ and $\delta > 0$, we can compute a $(1+\varepsilon, 1+\delta)$-approximate answer to a rectilinear k-center query in $O^*((k/\varepsilon)\log n + ((k^3/\delta)\log n)^4 + (k/\varepsilon^2 + (k^3/\delta))^{2k+2})$ time, where the O^*-notation hides $O(\text{polylog}(k/\delta))$ factors.*

Note that for constant k and $\varepsilon = \delta$ the query time simplifies to $O^*((1/\varepsilon^4)\log^4 n + (1/\varepsilon)^{4k+4})$. Also note that the time bound stated in the theorem only includes the time to compute the set of squares defining the clustering. If we want to also report an appropriate assignment of points to the squares, we have to add an $O(k^2 + |S_Q| \log |S_Q|)$ term; see Lemma 6.

Remark. The algorithm can be generalized to the rectilinear k-center problem in higher dimensions, and to the Euclidean k-center problem; we only need to plug in an appropriate appropriate δ-approximation algorithm and an appropriate algorithm for the 0/1-weighted version of the problem.

4 Exact k-Center Queries in $\mathbb{R}^1$

In this section we consider k-center queries in $\mathbb{R}^1$. Here we are given a set S of n points in $\mathbb{R}^1$ that we wish to preprocess into a data structure such that, given a query interval Q and a natural number $k \geqslant 2$, we can compute a set $\mathcal{C}$ of at most k intervals of the same length that together cover all points in $S_Q := S \cap Q$ and whose length is minimum. We obtain the following result.

Theorem 3. *Let S be a set of n points in $\mathbb{R}^1$. There is a data structure that uses $O(n)$ storage such that, for a query range Q and query value $k \geqslant 2$, we can answer a rectilinear k-center query in $O(\min(k^2 \log^2 n, 3^k \log n))$ time.*

The rest of the section is dedicated to the proof of the theorem. Our data structure is simply a sorted array on the points in S and therefore it needs only $O(n)$ space, but it has two different query algorithms. We call the query algorithms *a query algorithm for large k* and *a query algorithm for small k*. (See Section 4.1 and Section 4.2.) Both query algorithms start by shrinking the query interval Q such that its left and right endpoints coincide with a point in S_Q. This can obviously be done in $O(\log n)$ time. With a slight abuse of notation we still denote the shrunk interval by Q. Let x, x' be its left and right endpoints, respectively, so $Q = [x, x']$.

4.1 A Query Algorithm for Large k

This query algorithm uses a subroutine Decider which, given an interval Q', a length L and integer $\ell \leqslant k$, can decide in $O(\ell \log n)$ time if all points in $S \cap Q'$ can be covered by ℓ intervals of length L. The global query algorithm then performs a binary search, using Decider as subroutine, to find a pair of points $p_i, p_{i+1} \in S_Q$ such that the first interval in an optimal solution covers p_i but not p_{i+1}. Then an optimal solution is found recursively for $k-1$ clusters within the query interval $Q \cap [p_{i+1}, \infty)$. Next we describe the procedure Decider.

The Decider*-procedure.* The procedure Decider takes as input an integer ℓ, a number L, and an interval $Q' = [a, a']$. It returns yes if Q' can be covered by at most ℓ subintervals of length L, and no otherwise. Decider works as follows. Use binary search to find the first point $p_i \in S \cap Q'$ not covered by the interval $[a : a + L]$, set $a := p_i$ and recurse. This continues until either all points in $S \cap Q'$ are covered, or more than ℓ intervals are used. The Decider runs in $O(\ell \cdot \log n)$ time and outputs yes in the first case and outputs no in the latter case.

The global query algorithm. Given $Q := [x, x']$ and an integer k, we handle a query as follows. Let $S_Q := \{p_i, \ldots, p_j\}$, where the points are numbered from left to right. Thus $x = p_i$ and $x' = p_j$. We do a binary search on $\{p_i, \ldots, p_j\}$ to find the smallest index i^* with $i \leqslant i^* \leqslant j$ such that S_Q can be covered by k intervals of length $L := p_{i^*} - x$. Each decision in the binary search takes $O(k \log n)$ time by a call to Decider, so the entire binary search takes $O(k \log^2 n)$ time.

Let $\mathrm{Opt}_k(P)$ denote the minimum interval length needed to cover the points in a set P by k intervals. After finding i^* we know that

$$p_{i^*-1} - x < \mathrm{Opt}_k(S_Q) \leqslant p_{i^*} - x.$$

If $\mathrm{Opt}_k(S_Q) < p_{i^*} - x$, then the first interval in an optimal solution covers $\{p_i, \ldots, p_{i^*-1}\}$ and the remaining intervals cover $\{p_{i^*}, \ldots, p_j\}$. Now we recursively compute $\mathrm{Opt}_{k-1}(\{p_{i^*}, \ldots, p_j\})$, and since

$$p_{i^*-1} - x < \mathrm{Opt}_{k-1}(\{p_{i^*}, \ldots, p_j\}) \leqslant p_{i^*} - x,$$

we can safely report $\mathrm{Opt}_k(S_Q) = \mathrm{Opt}_{k-1}(\{p_{i^*}, \ldots, p_j\})$.

It remains to analyze the running time of a query. The binary search takes $O(k \log^2 n)$ times, after which we do a recursive call in which the value of k has decreased by 1. (The problem is easily solved in $O(\log n)$ time when $k = 1$.) Hence the number of recursive calls is k, leading to an $O(k^2 \log^2 n)$ query time, as claimed. Finding an optimal solution—and not just the value of an optimal solution—can be done within the same time bound. We get the following lemma.

Lemma 10. *Let S be a set of n points in $\mathbb{R}^1$. There is a data structure that uses $O(n)$ storage such that, for a query range Q and query value $k \geqslant 2$, we can answer a rectilinear k-center query in $O(k^2 \log^2 n)$ time.*

4.2 A Query Algorithm for Small k

Here we present the second query algorithm of the data structure, which is more efficient for small values of k. We begin with the following definition.

Definition 3. *Let S_Q be a set of points inside a query interval $Q = [x, x']$, such that $x, x' \in S_Q$. We call a point $r \in Q$ a* fair split point *if there is an optimal solution $\mathcal{C}_{opt}(Q) := \{I_1, I_2, \dots, I_k\}$ for the k-center problem on S_Q such that*

(i) r does not lie in the interior of any interval $I_j \in \mathcal{C}_{opt}(Q)$, and
(ii) the number of intervals in $\mathcal{C}_{opt}(Q)$ lying to the left of r is $k(r - x)/(x' - x)$.

Note that the split point r is not necessarily a point in S_Q, that is, it is not one of the given points. The following lemma is crucial in our analysis.

Lemma 11. *Let* $\mathrm{Split}(Q) := \{s_1, s_2, \dots, s_{k-1}\}$ *denote the set of points that partition Q into k equal-size subintervals. Then at least one of the points of* $\mathrm{Split}(Q)$ *is a fair split point.*

Proof. First we prove that there exists a point in $\mathrm{Split}(Q)$ that does not lie in the interior of some $I_j \in \mathcal{C}_{\mathrm{opt}}(Q)$. To this end, we observe that if the length of optimal intervals equals $(x' - x)/k$, then the optimal solution is equal to the subdivision of Q defined by the split points, and so the lemma trivially holds. Otherwise, the length of optimal intervals is strictly smaller than $(x' - x)/k$. But then an interval in $\mathcal{C}_{\mathrm{opt}}(Q)$ can contain at most one point from $\{s_0, \dots s_k\}$, where $s_0 := x$ and $s_k := x'$. Since s_0 and s_k are points in S_Q, there is an interval in $\mathcal{C}_{\mathrm{opt}}(Q)$ containing s_0 and one containing s_k. Hence, the remaining $k - 2$ intervals in $\mathcal{C}_{\mathrm{opt}}(Q)$ can cover at most $k - 2$ points from the split points $\{s_1, \dots s_{k-1}\}$ and so at least one of the split points will not be covered by the union of the subintervals.

It remains to prove that for at least one of the points of $\mathrm{Split}(Q)$ that satisfies Condition (i) in Definition 3, it also satisfies Condition (ii) in Definition 3. First consider the case that there is only one s_i with $0 < i < k$ that is not the interior of any I_j. Let $\ell_i := |\{I_j : I_j \subset [s_0, s_i]\}|$ denote the number of intervals to the left of s_i, and let and $f_i := \ell_i - i$. Since all the s_j with $s_0 \leqslant s_j < s_i$ are contained in distinct intervals from $\mathcal{C}_{\mathrm{opt}}(Q)$, we have $f_i \geqslant 0$. But since the same holds for all s_j with $s_i < s_j \leqslant s_k$, the number of intervals to the right of s_i is at least $k - i$. Hence, $f_i \leqslant k - (k - i) - i = 0$. We conclude that $f_i = 0$, so s_i is a fair split point.

Next we consider the case that several s_i are not in the interior of any I_j. Let $0 < i_1 < \dots < i_m < k$ be the corresponding indices. By the same arguments as above we have $f_{i_1} \geqslant 0$ and $f_{i_m} \leqslant 0$. Furthermore the sequence ℓ_i is non-decreasing, which implies $f_{i_j+1} \geqslant f_{i_j} - 1$. As a consequence, there is an i_j with $f_{i_j} = 0$. It follows that s_{i_j} is a fair split point.

Lemma 11 suggests the following approach. Again, the data structure is just a sorted array on the points in S. A query with range $Q = [x, x']$ and parameter k is answered as follows. Search the array for the successor $s(x)$ of x and the predecessor $p(x')$ of x' in S. Replace Q with $[s(x), p(x')]$, so that the left and right endpoints of the modified range Q are points from S. Partition Q into k equal-size subintervals. At each split point s_i of Q, recursively solve the problem on $Q_{\mathrm{left}} := [x, s_i]$

with parameter $k_{\mathrm{left}} := i$ and on $Q_{\mathrm{left}} := [s_i, x']$ with parameter $k_{\mathrm{right}} := k - i$. By Lemma 11, at (at least) one of the split points of Q the union of the returned intervals is an optimal solution. Moreover, we can easily maintain the best solution as we try all split points, so that after trying all split points we can return an optimal solution.

The recursion ends when $k = 1$. In this case we report $[s(x), p(x')]$ as the optimal solution. We obtain the following result.

Lemma 12. *Let S be a set of n points in $\mathbb{R}^1$. There is a data structure that uses $O(n)$ storage such that, for a query range Q and query value $k \geqslant 2$, we can answer a rectilinear k-center query in $O(3^k \log n)$ time.*

Proof. It takes $O(\log n)$ time to find the successor and the predecessor of x and x' in S. Hence, we obtain the following recurrence for the time $T(k, n)$ needed to answer a k-center query on a point set of size n:

$$T(k,n) \;\leqslant\; \begin{cases} O(\log n) & \text{if } k = 1 \\ O(\log n) + \sum_{i=1}^{k-1} T(i,n) + T(k-i,n) & \text{if } k > 1 \end{cases}$$

which solves to $T(n, k) = O(3^k \log n)$. To see this, note that the for recurrence

$$T^*(k) = \sum_{i=1}^{k-1} T^*(i) + T^*(k-i)$$

we have

$$T^*(k) = 2\sum_{i=1}^{k-1} T^*(i) = 3T^*(k-1),$$

so with $T^*(1) = 1$ we obtain $T^*(k) = 3^{k-1}$, which implies $T(n, k) = O(3^k \log n)$.

5 Exact Rectilinear 2- and 3-Center Queries in $\mathbb{R}^2$

Suppose we are given a set $S = \{p_1, p_2, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ and an integer k. In this section we build a data structure $\mathcal{D}$ that stores the set S and, given an orthogonal query rectangle Q, can be used to quickly find an optimal solution for the k-center problem on $S_Q := S \cap Q$ for $k = 2$ or 3.

5.1 2-Center Queries

We begin by a quick overview of our approach. We start by shrinking the query range Q such that each edge of Q touches at least one point of S. (The time for this step is subsumed by the time for the rest of the procedure.) It is well known that if we want to cover S_Q by two squares σ, σ' of minimum size, then σ and σ' both share a corner with Q and these corners are opposite corners of Q. We say that σ and σ' are *anchored* at the corner they share with Q. Thus we need to find optimal solutions for the two cases—σ and σ' are anchored at the topleft and bottomright corner of Q, or at the topright and bottomleft corner—and return the better one. Let c and c' be the topleft and the bottomright corners of Q. In the following we describe how to compute two squares σ and σ' of minimum size that are anchored at c and c', respectively, and whose union covers S_Q. The topright/bottomleft case can then be handled in the same way.

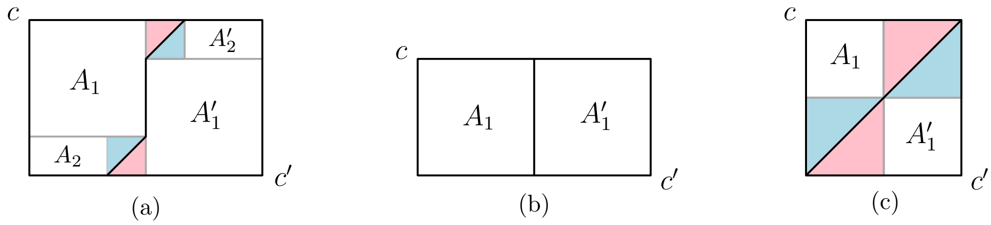

Fig. 1: Various types of L_∞-bisectors. The bisectors are shown in blue. (a): Q is "fat". The regions A_j, A'_j for $j = 1, 2$ are shown with text. (b): Q is "thin". The regions A_j and A'_j for $j = 2, 3, 4$ are empty. (c): Q is a square. The regions A_j and A'_j for $j = 2$ are empty. In both (a) and (c) regions A_3, A'_3 are colored in blue and A_4, A'_4 are colored in red.

First we determine the L_∞-bisector of c and c' inside Q; see Figure 1. The bisector partitions Q into regions A and A', that respectively have c and c' on their boundary. Obviously in an optimal solution (of the type we are focusing on), the square σ must cover $S_Q \cap A$ and the square σ' must cover $S_Q \cap A'$. To compute σ and σ', we thus need to find the points $q \in A$ and $q' \in A'$ with maximum L_∞-distance to the corners c and c', respectively. To this end, we partition A and A' into subregions such that in each of the subregions the point with maximum L_∞-distance to its corresponding corner can be found quickly via appropriate data structures discussed below. We assume w.l.o.g. that the x-span of Q is at least its y-span. We begin by presenting the details of such a partitioning for Case (a) of Figure 1—Case (b) and Case (c) can be seen as special cases of Case (a).

As Figure 1 suggests, we partition A and A' into subregions. We denote these subregions by A_j and A'_j, for $1 \leqslant j \leqslant 4$. From now on we focus on reporting the point $q \in S$ in A with maximum L_∞-distance to c; finding the furthest point from c' inside A' can be done similarly. Define four points $p(A_j) \in S$ for $1 \leqslant j \leqslant 4$ as follows.

- The point $p(A_1)$ is the point of S_Q with maximum L_∞-distance to c in A_1. Note that this is either the point with maximum x-coordinate in A_1 or the point with minimum y-coordinate.
- The point $p(A_2)$ is a bottommost point in A_2.
- The point $p(A_3)$ is a bottommost point in A_3.
- The point $p(A_4)$ is a rightmost point in A_4.

Clearly
$$q = \arg \max_{1 \leqslant j \leqslant 4} \{d_\infty(p(A_j), c)\}, \tag{3}$$
where $d_\infty(.)$ denotes the L_∞-distance function.

Data structure. Our data structure now consists of the following components.

- We store S in a data structure $\mathcal{D}_1$ that allows us to report the extreme points in the x-direction and in the y-direction inside a rectangular query range. For this we use the structure by Chazelle [9], which needs $O(n \log^\delta n)$ space and has $O(\log n)$ query time, where $\delta > 0$ is an arbitrary small (but fixed) constant.
- We store S in a data structure $\mathcal{D}_2$ with two components. The first component should answer the following queries: given a 45° query cone whose top bounding line is horizontal and that

is directed to the left—we obtain such a cone when we extend the region A_4 into an infinite cone—, report the rightmost point inside the cone. The second component should answer similar queries for cones that are the extension of A_3.

Lemma 13 proves the existence of a linear-size data structure that implements such a component and that has $O(\log n)$ query time.

Lemma 13. *Each component of $\mathcal{D}_2$ has complexity $O(n)$ and it can be built in $O(n \log n)$ time.*

Proof. We describe the component for the following queries: given a 45° query cone whose bottom bounding line is horizontal and that is directed to the right, report the leftmost point inside the cone. Our structure for such queries is defined as follows. For each point $p_i \in p$ consider the inverted cone with apex p_i, that is, the 45° cone whose top edge is horizontal. We now add these inverted cones from right to left, where we add each cone "on top of" the existing cones. This gives us a linear-size subdivision, which is a Voronoi diagram for the distance function induced by our problem, which we preprocess for point location. If we then do a point-location query in the subdivision with the apex of our query cone, then this tells us the leftmost point inside the query cone.

To construct the structure, we use a sweep-line approach. The sweep line is a vertical line that moves from right to left. The sweep line halts at each point $p_i \in S$, and computes the Voronoi cell of p_i, denoted with $\mathrm{Vor}(p_i)$, as the set of all the points in the plane that lie in the unbounded left 45°-cone starting at p_i. If $\mathrm{Vor}(p_i)$ intersects $\mathrm{Vor}(p_j)$, for some $j < i$, then the region $\mathrm{Vor}(p_i) \subseteq \mathrm{Vor}(p_j)$ will belong to $\mathrm{Vor}(p_i)$. Observe that $\mathrm{Vor}(p_i)$ can intersect at most one $\mathrm{Vor}(p_j)$ with $j < i$ and therefore updating $\mathrm{Vor}(p_i)$ can be done easily. See Figure 2 for a picture of execution of the algorithm for a few successive iterations.

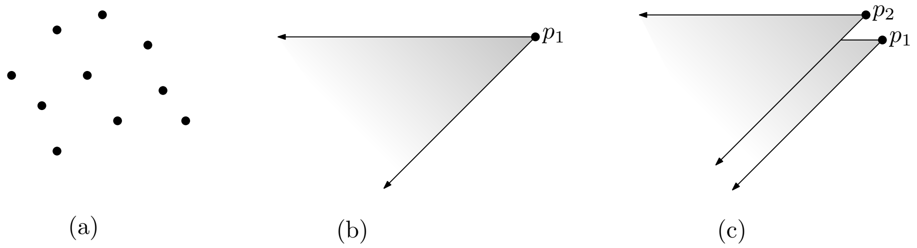

Fig. 2: A point set S and the Voronoi cells of the first two points of S visited by the sweep-line algorithm described in Lemma 13.

Query procedure. Given an axis-aligned query rectangle Q, we first (as already mentioned) shrink the query range so that each edge of Q contains at least on point of S. Then compute the L_∞-bisector of Q. Query $\mathcal{D}_1$ with A_1 and A_2, respectively, to get the points $p(A_1)$ and $p(A_2)$. Then query $\mathcal{D}_2$ with u and u' to get the points $p(A_3)$ and $p(A_4)$, where u and u' are respectively the bottom and the top intersection points of L_∞-bisector of Q and the boundary of Q. Among the at most four reported points, take the one with maximum L_∞-distance the corner c. This is the point $q \in S_Q \cap A$ furthest from c.

Compute the point $q' \in S_Q \cap A$ furthest from c' in a similar fashion. Finally, report two minimum-size congruent squares σ and σ' anchored at c and c' and containing q and q', respectively.

Putting everything together, we end up with the following theorem.

Theorem 4. *Let S be a set of n points in the plane. For any fixed $\delta > 0$, there is a data structure using $O(n \log^\delta n)$ space that can answer rectilinear 2-center queries in $O(\log n)$ time.*

Remark. We note that the query time in Theorem 4 can be improved in the word-RAM model to $O(\log \log n)$ by using the range successor data structure of Zhou [24], and the point-location data structure for orthogonal subdivisions by de Berg *et al.* [11].

5.2 3-Center Queries

Given a (shrunk) query range Q, we need to compute a set $\{\sigma, \sigma', \sigma''\}$ of (at most) three congruent squares of minimal size whose union covers S_Q. It is easy to verify (and is well-known) that at least one of the squares in an optimal solution must be anchored at one of the corners of Q. Hence and w.l.o.g. we assume that σ is anchored at one of the corners of Q. We try placing σ in each corner of Q and select the placement resulting in the best overall solution. Next we briefly explain how to find the best solution subject to placing σ in the leftbottom corner of Q. The other cases are symmetric. We perform two separate binary searches; one will test placements of σ such that its right side has the same x-coordinate as a point in S, the other will be on possible y-coordinates for the top side. During each of the binary searches, we compute the smallest axis-parallel rectangle $Q' \subseteq Q$ containing the points of $Q \backslash \sigma$ (by covering $Q \backslash \sigma$ with axis-aligned rectangles and querying for extreme points in these rectangles). We then run the algorithm for $k = 2$ on Q'. We need to ensure that this query ignores the points already covered by σ. For this, recall that for $k = 2$ we covered the regions A and A' by suitable rectangular and triangular ranges. We can now do the same, but we cover $A \setminus \sigma$ and $A' \setminus \sigma$ instead.

After the query on Q', we compare the size of the resulting squares with the size of σ to guide the binary search. The process stops as soon as the three sizes are the same or no further progress in the binary search can be made.

Putting everything together, we end up with the following theorem.

Theorem 5. *Let S be a set of n points in the plane. For any fixed $\delta > 0$, there is a data structure using $O(n \log^\delta n)$ space that can answer rectilinear 3-center queries in $O(\log^2 n)$ time.*

Remark. Similar to Theorem 4, the query time in Theorem 5 can be improved in the word-RAM model of computation to $O(\log n \log \log n)$ time.

6 Discussion

In this paper we presented a general method to preprocess a given point set S in $\mathbb{R}^d$ into a data structure for fast range-clustering queries on the subset of S that lies inside a given axis-aligned query box Q. Our main result is a general method to compute a $(1 + \varepsilon)$-approximation to a range-clustering query, where $\varepsilon > 0$ is a parameter that can be specified as part of the query.

Our method applies to a large class of clustering problems, including k-center clustering in any L_p-metric and a variant of k-center clustering where the goal is to minimize the sum (instead of maximum) of the cluster sizes. We also extended our method to deal with capacitated k-clustering problems, where each cluster should contain at most a given number of points. For the special cases of rectilinear k-center clustering in $\mathbb{R}^1$ and in $\mathbb{R}^2$ for $k = 2$ or 3, we described data structures that answer range-clustering queries exactly.

We close the paper by stating the following open questions.

- Can the bound in Theorem 1 (and the bounds in Corollary 1) be improved?
- Is it possible to design efficient exact data structures for rectilinear k-center queries when $k > 3$?
- Can any of the data structures presented in this paper be made dynamic?
- Is it possible to extend our results on approximate queries to non-regular cost functions (for example, for the k-means problem)?

Acknowledgements. This research was initiated when the first author visited the Department of Computer Science at TU Eindhoven during the winter 2015–2016. He wishes to express his gratitude to the other authors and the department for their hospitality. The last author wishes to thank Timothy Chan for valuable discussions about the problems studied in this paper.

References

1. M. Abam, P. Carmi, M. Farshi, and M. Smid. On the power of the semi-separated pair decomposition. *Compututational Geometry: Theory and Applications*, 46:631–639, 2013.
2. P. K. Agarwal, R. B. Avraham, and M. Sharir. The 2-center problem in three dimensions. *Compututational Geometry: Theory and Applications*, 46:734–746, 2013.
3. P. K. Agarwal and C. M. Procopiuc. Exact and approximation algorithms for clustering. *Algorithmica*, 33:201–226, 2002.
4. S. Arya, D. M. Mount, and E. Park. Approximate geometric MST range queries. In *Proc. 36th International Symposium on Computational Geometry (SoCG)*, pages 781–795, 2015.
5. P. Brass, C. Knauer, C. Shin, M. H. M. Smid, and I. Vigan. Range-aggregate queries for geometric extent problems. In *Computing: The Australasian Theory Symposium 2013, CATS'13*, pages 3–10, 2013.
6. V. Capoyleas, G. Rote, and G. Woeginger. Geometric clusterings. *Journal of Algorithms*, 12:341–356, 1991.
7. T. M. Chan. Geometric applications of a randomized optimization technique. *Discrete & Compututational Geometry*, 22:547–567, 1999.
8. T. M. Chan. More planar two-center algorithms. *Compututational Geometry: Theory and Applications*, 13:189–198, 1999.
9. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17:427–462, 1988.
10. A. W. Das, P. Gupta, K. Kothapalli, and K. Srinathan. On reporting the L_1-metric closest pair in a query rectangle. *Information Processing Letters*, 114:256–263, 2014.
11. M. de Berg, M. van Kreveld, and J. Snoeyink. Two- and three-dimensional point location in rectangular subdivisions. *Journal of Algorithms*, 18:256–277, 1995.
12. D. Eppstein. Faster construction of planar two-centers. In *Proc. 8th Annual ACM-SIAM Symposiun on Discrete Algorithms (SODA)*, pages 131–138, 1997.
13. P. Gupta, R. Janardan, Y. Kumar, and M. Smid. Data structures for range-aggregate extent queries. *Compututational Geometry: Theory and Applications*, 47:329–347, 2014.

14. S. Har-Peled. *Geometric Approximation Algorithms*, volume 173 of *Mathematical surveys and monographs*. American Mathematical Society, 2011.
15. S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *Proc. 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–300, 2004.
16. M. Hoffmann. A simple linear algorithm for computing rectilinear 3-centers. *Compututational Geometry: Theory and Applications*, 31:150–165, 2005.
17. R. Z. Hwang, R. Lee, and R. C. Chang. The generalized searching over separators strategy to solve some NP-hard problems in subexponential time. *Algorithmica*, 9:398–423, 1993.
18. S. Khare, J. Agarwal, N. Moidu, and K. Srinathan. Improved bounds for smallest enclosing disk range queries. In *Proc. 26th Canadian Conference on Computational Geometry (CCCG)*, 2014.
19. H. P. Lenhof and M. H. M. Smid. Using persistent data structures for adding range restrictions to searching problems. *Theoretical Informatics and Applications*, 28:25–49, 1994.
20. Y. Nekrich and M. H. M. Smid. Approximating range-aggregate queries using coresets. In *Proc. 22nd Canadian Conference on Computational Geometry (CCCG)*, pages 253–256, 2010.
21. J. M. Phillips. Algorithms for ε-approximations of terrains. In *Proc. 35th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 447–458, 2008.
22. M. Sharir. A near-linear time algorithm for the planar 2-center problem. *Discrete & Compututational Geometry*, 18:125–134, 1997.
23. M. Sharir and E. Welzl. Rectilinear and polygonal p-piercing and p-center problems. In *Proc. 12th International Symposium on Computational Geometry (SoCG)*, pages 122–132, 1996.
24. G. Zhou. Two-dimensional range successor in optimal time and almost linear space. *Information Processing Letters*, 116:171–174, 2016.