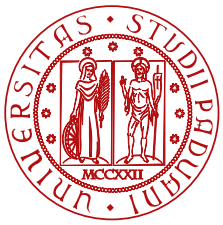

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

UNIVERSITY OF PADUA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN COMPUTER ENGINEERING

Dynamic k-Center Clustering with Lifetimes

Supervisor
Prof. Pietracaprina Andrea

Candidate
Moretti Simone

Co-supervisor
Prof. Pucci Geppino

Student ID
2139223

ANNO ACCADEMICO 2025-2026

Data di laurea 07/07/2026

Abstract

The k-center problem is a fundamental clustering variant with applications in learning systems and data summarization. In several real-world scenarios, the dataset to be clustered is not static, but evolves over time, as new data points arrive and old ones become stale. To account for dynamicity, the k-center problem has been mainly studied under the sliding window setting, where only the N most recent points are considered non-stale, or the fully dynamic setting, where arbitrary sequences of point arrivals and deletions without prior notice may occur. In this thesis, we introduce the dynamic setting with lifetimes, which bridges the two aforementioned classical settings by still allowing arbitrary arrivals and deletions, but making the deletion time of each point known upon its arrival. Under this new setting, we devise a deterministic $(2+\varepsilon)$-approximation algorithm with $O(k/\varepsilon)$ amortized update time and memory usage linear in the number of currently active points. Moreover, we develop a deterministic $(6+\varepsilon)$-approximation algorithm that, under "tame" update sequences, has $O(k/\varepsilon)$ worst-case update time and heavily sublinear working memory.

Contents

Introduction

Clustering is one of the most basic and widely used tools for data analysis. Given a collection of objects with a notion of distance measuring how dissimilar they are, a clustering algorithm groups the objects so that similar ones end up together and dissimilar ones get grouped apart. This shows structure and produces useful information of an otherwise useless dataset. Among the many formalizations of this intuition, the *k-center* problem occupies a prominent place: given a set of points in a metric space and a number k, it asks to find k points, called *centers*, that minimize the maximum distance from any point to its closest center. It is a natural primitive in facility location, data summarization, coreset construction, feature selection, outlier detection and it is frequently used as a building block inside larger learning pipelines.

The problem is NP-hard, and, unless $P = NP$, no polynomial-time algorithm can approximate its optimal radius within a factor smaller than 2 [1]. This lower bound is matched by a remarkably simple greedy algorithm due to Gonzalez [1], which repeatedly selects as the next center the point farthest from those chosen so far and attains a 2-approximation in $O(nk)$ time.

Furthermore, in a growing number of applications, the dataset to be clustered is not fixed, but *evolves over time*. A series of *insertions* and *deletions* of points are provided in input, along with *queries* that ask for a clustering of the current *active* set of points at time t: X_t. Recomputing a clustering from scratch for each query is too expensive, so we seek *dynamic* algorithms that maintain a good solution incrementally, updating a small amount of internal state per operation and spending time and memory that scale with the active set rather than with the entire history of the data.

Insertions and deletions can be very structured or arbitrary, giving rise to two opposed models, well analyzed in the literature. In the *sliding-window* model, a point is considered active only as long as it is among the N most recent arrivals: expiration is completely determined by arrival order, so points leave in exactly the order in which they entered. Cohen-Addad et al. [2] obtained a $(6 + \varepsilon)$-approximation for k-center in sliding windows

saving only $O((k/\varepsilon)\log\Delta)$ points, obtaining a working memory *independent* of the size of the active set, where Δ denotes the aspect ratio of the data. At the other extreme lies the *fully dynamic* model, where insertions and deletions are arbitrary and not known in advance. This is the most general and demanding setting. Chan et al. [3] gave the first fully dynamic algorithm for k-center, and Bateni et al. [4] later refined the approach into the current state of the art, maintaining a $(2+\varepsilon)$-approximation with total running time $O\big(n \cdot k\frac{\log\Delta}{\varepsilon}\log n\big)$ over a sequence of n operations.

In many real scenarios the assumption of unknown deletion times is far too pessimistic. Cached entries and DNS records carry an explicit time-to-live; reservations, licenses, and sessions come with an expiry date; a check-in in a location-based social network remains the user's current position exactly until their next check-in; and buffered items in packet routers or manufacturing pipelines are consumed within a bounded delay after being produced. The sliding-window model does assume predictable expiration, but only in the most rigid possible form, forcing every point to survive for exactly the same span and to expire in strict arrival order.

The dynamic setting with lifetimes. Motivated by this gap, this thesis studies k-center in the *dynamic setting with lifetimes*, which we position as a bridge between the two classical models. As in the fully dynamic setting, arrivals and deletions are arbitrary; but, as in the sliding-window setting, expirations are known in advance. We investigate this highly unexplored region of the literature to find algorithms that beat the state-of-the-art in update time or approximation efficiency.

Contributions

The contributions of this thesis are the following:

- We introduce and formalize the dynamic setting with lifetimes for the metric k-center problem, and we show how it relates to the sliding-window and fully dynamic settings.
- We devise a deterministic $(2+\varepsilon)$-approximation algorithm for this setting. We build on the structure of the fully dynamic algorithm of Chan et al. [3], but exploiting the known deletion times, achieving an amortized update time of $O((\log\Delta/\varepsilon)\,k)$, thereby removing the multiplicative $\log n$ factor incurred by the state-of-the-art fully dynamic algorithm of Bateni et al. [4], while using memory linear in the size

of the active set. The resulting update time is optimal up to a $O(\log \Delta/\varepsilon)$ factor, matching the $\Omega(|X|\, k)$ lower bound of [4] which holds even on offline algorithms.

- As a by-product, we show that dropping the knowledge of deletion times from this algorithm yields a new, purely fully dynamic $(2+\varepsilon)$-approximation. We conjecture that it retains the same expected update time and thus would improve on the fully dynamic state of the art, and, although we leave the analysis open, we include the variant in our experimental evaluation.

- We devise a deterministic $(6+\varepsilon)$-approximation algorithm with *sublinear* working memory. Extending the sliding-window construction of Cohen-Addad et al. [2] to arbitrary, known deletion times, the algorithm attains $O((\log \Delta/\varepsilon)\, k)$ *worst-case* update time and working memory independent of the active-set size, whenever the update sequence is "tame". Tameness is captured by a novel parameter, H-orderedness, that quantifies how far deletions may depart from arrival order. To the best of our knowledge, this is the first $(6+\varepsilon)$-approximation for k-center that operates outside the sliding-window model.

- We complement the theory with an experimental study. We implement our algorithms alongside those of Chan et al. [3] and Bateni et al. [4], and evaluate them on synthetic data and on the real-world Gowalla check-in dataset, comparing approximation quality, update and query times, and memory usage.

Structure of the thesis

The remainder of the thesis is organized as follows. Chapter 1 introduces metric spaces and the k-center problem, recalls the hardness of the problem and Gonzalez' greedy algorithm, and formalizes the dynamic setting together with the fully dynamic and lifetimes models. Chapter 2 is devoted to minimizing the approximation ratio: it reviews the fully dynamic algorithms of Chan et al. [3] and Bateni et al. [4], presents our $(2+\varepsilon)$-approximation with lifetimes and its analysis, and discusses the fully dynamic variant obtained by discarding deletion times. Chapter 3 turns to memory efficiency: it recalls the sliding-window algorithm of Cohen-Addad et al. [2], introduces the notion of H-orderedness, and develops and analyzes our sublinear-memory $(6+\varepsilon)$-approximation. Chapter 4 reports the experimental evaluation and draws conclusions.

Chapter 1

Preliminaries

In this chapter we provide the basic definitions needed to understand the subsequent work. We first introduce metric spaces, which give the notion of distance our clustering relies on, and then formalize the k-center problem in terms of this distance. Finally, we describe the dynamic setting in which our algorithms operate.

1.1 Clustering

Definition 1.1 (Metric Space)**.** *Let $\mathcal{X}$ be a set and $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ a function with the following properties:*

1. $d(x,y) = 0 \iff x = y \quad \forall x,y \in \mathcal{X}$;
2. $d(x,y) = d(y,x) \quad \forall x,y \in \mathcal{X}$ *(Symmetry);*
3. $d(x,y) \leq d(x,z) + d(z,y) \quad \forall x,y,z \in \mathcal{X}$ *(Triangle Inequality).*

*Then the pair $(\mathcal{X},d)$ is called a **metric space**, and d is called the **distance**. An element $x \in \mathcal{X}$ is called a **point**.*

Throughout, we let $d_{\min}$ and $d_{\max}$ denote the minimum and maximum distances between any two distinct points of X, and define the **aspect ratio** of X as $\Delta = d_{\max}/d_{\min}$. As is standard in most previous works, our algorithms assume that $d_{\min}$ and $d_{\max}$ are known in advance.

The distance gives an indication of the similarity between points: the smaller the distance between two points, the more similar they are. Finding a meaningful notion of clustering in a fully general setting is notoriously elusive, so we restrict our attention to

center-based clustering, in which each cluster is represented by one of its points, called a *center*, and every point is assigned to the center it is closest to. To make this precise, we first extend the distance from points to sets.

Definition 1.2 (Point-to-set distance and radius)**.** *Let $X \subseteq \mathcal{X}$ be a finite set of points. For a point $p \in X$ and a subset $C \subseteq X$, the distance of p from C is*

$$d(p, C) := \min_{q \in C} d(p, q).$$

*For any $S \subseteq X$, the **radius** of C with respect to S is*

$$r_C(S) := \max_{p \in S} d(p, C).$$

A set of centers $C \subseteq X$ induces a partition of X into $|C|$ *clusters*, obtained by assigning each point to its closest center (with ties broken arbitrarily). With this assignment, $r_C(X)$ is the largest distance between any point and the center of its cluster, and thus measures how tight the induced clustering is: the smaller the radius, the more compact the clusters. The k-center problem asks for the choice of at most k centers that makes this radius as small as possible.

1.1.1 k-Center Clustering

Definition 1.3 (k-Center Clustering)**.** *Given a finite set $X \subseteq \mathcal{X}$ and an integer $k \in \mathbb{N}_{>0}$, the k-**center** problem requires to find a subset $C \subseteq X$ with $|C| \leq k$ that minimizes the radius $r_C(X)$, i.e.,*

$$C^* = \operatorname*{arg\,min}_{C \subseteq X,\, |C| \leq k} r_C(X).$$

*The points of C are called **centers**, and we denote the radius of the optimal solution by $r_k^*(X) = r_{C^*}(X)$.*

Observation 1. *To specify a solution it suffices to provide the set of centers C: the clusters are recovered by assigning each point to its closest center, i.e., cluster i is the set $\{x \in X \mid i = \arg\min_j \{d(x, c_j)\}\}$.*

We now present a simple brute-force algorithm for solving k-center clustering, which enumerates all possible sets of k centers and keeps the one of minimum radius.

Algorithm 1: k-CenterBruteforce(X, k)

```
opt ← ∞
c* ← (0,...,0)
C* ← (∅,...,∅)
for (c_1,...,c_k) ∈ X^k do
    t ← r_{c_1,...,c_k}(X)
    if opt > t then
        opt ← t
        c* ← (c_1,...,c_k)
        for 1 ≤ i ≤ k do
            C*_i ← {x ∈ X | i = arg min_j {d(x,c_j)}}
return (c*,C*)
```

Proposition 1.1. *The k-center clustering problem is NP-hard. In other words, it is impossible to find a polynomial time algorithm that computes an optimal solution for an arbitrary X, unless $P = NP$. Moreover, it is impossible to achieve an approximation factor $2 - \varepsilon$, for any fixed $\varepsilon > 0$, unless $P = NP$.*

Given the NP-hardness of the problem, approximate solutions are sought. One such approximation algorithm was given by Gonzalez [1]:

Algorithm 2: Gonzalez(X,k)

```
c ← (0,...,0)
c_1 ← arbitrary x ∈ X
for 1 < i ≤ k do
    c_i ← arg max_{x∈X} d(x,{c_1,...,c_{i-1}})
C ← (∅,...,∅)
for 1 ≤ i ≤ k do
    C_i ← {x ∈ X | i = arg min_j {d(x,c_j)}}
return (c,C)
```

The algorithm picks the centers iteratively, choosing each new center to be the point that is farthest from the set of centers selected so far.

Proposition 1.2. *Gonzalez' algorithm produces a 2-approximation of k-center in time $O(nk)$.*

1.2 Dynamic Algorithms

In some applications the pointset X is not fixed, but changes over time. We define the *dynamic setting* to model such cases. In the dynamic setting:

- X is the set of all points that ever appear during the whole process;
- X_t is the set of *active* points at time t;
- $\textsc{Insert}(p, t)$ denotes the insertion of a point in the active set at time t;
- $\textsc{Delete}(p, t)$ denotes the deletion of a point from the active set at time t;
- for a given point $p \in X$ we denote with $t_{\text{arr}}(p)$ its insertion (*arrival*) time and with $t_{\text{del}}(p)$ its deletion time;
- all insertion operations happen at distinct times (i.e., $p_1 = p_2 \iff t_{\text{arr}}(p_1) = t_{\text{arr}}(p_2)\ \forall p_1,p_2 \in X$), whereas deletions may share the same time;
- a point p is active during the time interval $[t_{\text{arr}}(p),t_{\text{del}}(p))$ (i.e., $p \in X_t \iff t_{\text{arr}}(p) \leq t < t_{\text{del}}(p)\ \forall p \in X$).

The goal of a dynamic algorithm is to maintain a solution for the problem on the active set X_t without recomputing it from scratch at each step, so that the cost of an update depends on the size of the active set X_t rather than on the size of the whole pointset X. For the k-center problem in particular, the algorithm should keep, at any time t, a set of at most k centers $C_t \subseteq X_t$ whose radius $r_{C_t}(X_t)$ is close to the optimal one $r_k^*(X_t)$, while keeping the time and memory it uses small.

The way in which deletions are handled distinguishes the two main dynamic settings that we consider.

Definition 1.4 (Fully dynamic setting)**.** *In the* ***fully dynamic setting****, insertions and deletions are arbitrary and issued by an external source: at any time, a new point may be inserted into the active set, or an active point may be deleted from it. The deletion time $t_{\text{del}}(p)$ of a point p is not known in advance, and is revealed only when p is actually deleted.*

Definition 1.5 (Dynamic setting with lifetimes)**.** *In the* ***dynamic setting with lifetimes****, whenever a point p is inserted at time $t_{\text{arr}}(p)$, its deletion time $t_{\text{del}}(p) > t_{\text{arr}}(p)$ is specified as well, thereby fixing the* lifetime $[t_{\text{arr}}(p),t_{\text{del}}(p))$ *of p at the moment of its arrival. We say that p* expires *at time $t_{\text{del}}(p)$, when it leaves the active set.*

The algorithms presented in this thesis operate in the setting with lifetimes, and crucially exploit the knowledge of the deletion times to obtain better guarantees than what is possible in the fully dynamic setting.

Chapter 2

Dynamic k-Center: Minimizing Approximation

In this chapter we will look at algorithms for maintaining a $(2+\varepsilon)$-approximation of the dynamic k-center. We will first look at 2 previous algorithms for the fully-dynamic case, then we will present our improvement in the lifetimes setting. The first two sections will describe respectively the work of Chan et al. [3] and of Bateni et al. [4]. The work of Bateni is the current state of the art for k-center approximation in the fully dynamic setting, achieving a $(2+\varepsilon)$-approximation in time $O\big(n \cdot k \frac{\log \Delta}{\varepsilon} \log n\big)$. We will show how the setting with lifetimes allows to remove the $\log n$ multiplicative factor from the complexity, achieving a nearly optimal running time. The work of Chan, whilst requiring a worse runtime of $\Omega\left(n \cdot k^2 \frac{\log \Delta}{\varepsilon}\right)$, is still presented as our algorithm is based on it.

All the 3 presented algorithms make a set of guesses Γ of the optimal radius and keep for each of them a suitable data structure. To better understand the power of guessing the radius we will state the following proposition:

Proposition 2.1. *Let $\gamma \in \Gamma$ be a guess of the optimal radius. If at time t $\exists x_1, \ldots, x_{k+1} \in X_t$ s.t. $d(x_i, x_j) > 2\gamma \ \forall i \neq j$, then $r_k^*(X_t) > \gamma$.*

Proof. Suppose that $r_k^*(X_t) \leq \gamma$. Then $\exists c_1, \ldots, c_k \in X_t$ centers with respective clusters $C_1, \ldots, C_k$ s.t. $d(x, c_i) \leq \gamma \ \forall 1 \leq i \leq k, x \in C_i$. By the pigeonhole principle, at least 2 of the $k+1$ x_i will be in the same cluster C_j. Assume wlog that $x_1, x_2 \in C_1$. Then we will have:

$$2\gamma < d(x_1, x_2) \leq d(x_1, c_1) + d(x_2, c_1) \leq \gamma + \gamma = 2\gamma$$

Where we used the triangle inequality.

We obtained that $2\gamma < 2\gamma$, which is absurd, meaning that $r_k^*(X_t) \leq \gamma$ must be false. □

In all 3 algorithms, for each guess $\gamma \in \Gamma$, we will at each time find either k centers with respective clusters at distance $\leq 2\gamma$ or $k+1$ points at pairwise distance $> 2\gamma$. We set:

$$\Gamma = \left\{(1+\beta)^i : \lfloor \log_{1+\beta} d_{\min} \rfloor \leq i \leq \lceil \log_{1+\beta} d_{\max} \rceil \right\}. \tag{2.1}$$

Proposition 2.2. *Let $\varepsilon > 0$. Suppose that Γ is as defined before with $\beta = \varepsilon/2$ and that for each γ we either have a clustering with radius 2γ or have $k+1$ points at distance $> 2\gamma$. Let $\gamma' \in \Gamma$ be the smallest $\gamma \in \Gamma$ s.t. we have a clustering of radius 2γ. $2\gamma'$ is a $(2+\varepsilon)$-approximation of the optimal radius.*

Proof. Given that for γ' we know k centers at distance $\leq 2\gamma'$ from all the other points we have that $r_k^*(X_t) \leq 2\gamma'$. It remains to prove that $r_k^*(X_t) \geq 2\gamma'/(2+\varepsilon)$.

Let i be s.t. $(1+\beta)^i = \gamma'$. If $i = \lfloor \log_{1+\beta} d_{\min} \rfloor$ then given that $d_{\min}$ is the minimal pairwise distance we have $r_k^*(X_t) \geq d_{\min}$ (unless $k = |X_t|$), therefore: $2\gamma'/(2+\varepsilon) < \gamma' \leq d_{\min} \leq r_k^*(X_t)$.

If $i > \lfloor \log_{1+\beta} d_{\min} \rfloor$ by the fact that γ' is minimal we know that a solution with radius $(1+\beta)^{i-1}$ is impossible, therefore:

$$r_k^*(X_t) > (1+\beta)^{i-1} = \frac{(1+\beta)^i}{1+\beta} = \frac{\gamma'}{1+\varepsilon/2} = \frac{2\gamma'}{2+\varepsilon}.$$

□

2.1 A $O(k^2)$ Fully Dynamic Algorithm

In this section we present the work of Chan et al. [3], who provided the first fully dynamic algorithm for the k-center problem. For ease of readability we will omit the subscript t indicating the time at which each object exists every time that it is not strictly necessary. Just remember that, being this a dynamic algorithm, all the presented data structure will change over time. Moreover, both proofs of the correctness and running time are omitted. The correctness is later proved in the analysis with lifetimes, the proofs for the running time can be found in the original paper.

As discussed before, the algorithm creates the set of guesses Γ and maintains a data structure $\mathcal{L}_\gamma$ for each of them. $\mathcal{L}^\gamma$ is composed by:

- A list $C^\gamma = (c_1^\gamma, \ldots, c_{\ell^\gamma}^\gamma)$ of $\ell^\gamma < k$ centers s.t. for all $x, y \in C^\gamma$, $d(x, y) > 2\gamma$.

- A list $D^\gamma = (D_1^\gamma, \ldots, D_{\ell^\gamma}^\gamma)$ of disjoint clusters s.t. $\forall 1 \leq i \leq \ell^\gamma, x \in D_i^\gamma$, $d(x, c_i^\gamma) \leq 2\gamma$.

- A set $U^\gamma = X \setminus \cup_{1 \leq i \leq \ell^\gamma} D_i^\gamma$ of unclustered points s.t. $\forall 1 \leq i \leq \ell^\gamma, x \in U^\gamma$, $d(c_i^\gamma, x) \geq 2\gamma$. Moreover, we require that $U^\gamma \neq \emptyset \implies \ell^\gamma = k$.

Having defined the data structure this way, it is quite simple to see that for each γ if $U^\gamma \neq \emptyset$ then there are at least $k+1$ points at distance $> 2\gamma$, otherwise a clustering with radius 2γ is given by (C^γ, D^γ).

The insertion procedure is quite simple:

Algorithm 3: INSERT(p, γ)

```
for i = 1, ... ℓ^γ do
    if d(p, c_i^γ) ≤ 2γ then
        Insert p in D_i^γ
        return
if ℓ^γ < k then
    Insert p in C^γ and in D_{ℓ^γ+1}^γ  ℓ^γ ← ℓ^γ + 1
else Insert p in U^γ;
```

It inserts p in the cluster of D^γ of lowest index whose center is at distance at most 2γ from p. If no such cluster exists, if D^γ comprises $\ell^\gamma < k$ clusters, then the procedure creates a new cluster by making p its center, otherwise, it inserts p in U^γ.

Deleting a point p is a bit trickier: if $x \notin C^\gamma$ it can be simply removed from its cluster without changing anything else. However if $x \in C^\gamma$ we need to do additional work to maintain the properties of our data structure. Let's look at the deletion procedure:

Algorithm 4: DELETE(p, γ)

if $p \notin C^\gamma$ **then**
  Remove p from its respective cluster in D^γ or from U^γ
  return
Let $1 \leq i \leq \ell^\gamma$ s.t. $p = c_i^\gamma$
$\hat{X} \leftarrow (\cup_{i \leq j \leq \ell^\gamma} D_i^\gamma) \cup U^\gamma$
$(\hat{C}, \hat{D}, \hat{U}) \leftarrow$ RANDRECLUST$(\hat{X}, \gamma, k - i + 1)$
$C^\gamma \leftarrow (c_1^\gamma, \ldots, c_{i-1}^\gamma) \oplus \hat{C}$
$D^\gamma \leftarrow (D_1^\gamma, \ldots, D_{i-1}^\gamma) \oplus \hat{D}$
$\ell^\gamma \leftarrow i + |\hat{C}| - 1$
$U^\gamma \leftarrow \hat{U}$

Where the $\oplus$ operator denotes the concatenation of 2 lists. As we can see if a point is not a center it gets simply deleted, otherwise all of the points that are in clusters with index $\geq$ than the index of the deleted center get reclustered. The reclustering procedure (Algorithm 5) takes this set in input, alongside a value κ, and returns up to κ clusters with corresponding centers picked at random from the set of points. The points left unclustered by this procedure, if any, are then put in U.

Algorithm 5: RANDRECLUST(X, γ, κ)

$U \leftarrow X$
$\ell \leftarrow 0$
while $U \neq \emptyset$ *and* $\ell < \kappa$ **do**
  $\ell \leftarrow \ell + 1$
  Pick c_ℓ from U uniformly at random
  $D_\ell \leftarrow \{x \in U | d(x, c_\ell) \leq 2\gamma\}$
  $U \leftarrow U \setminus D_\ell$
return $((c_1, \ldots, c_\ell), (D_1, \ldots, D_\ell), U)$

Theorem 2.1. *The algorithm described above obtains a $(2 + \varepsilon)$-approximation of the k-center Problem and runs in $O\big(n \cdot k^2 \frac{\log \Delta}{\varepsilon}\big)$ amortized time in expectation.*

2.2 The State-of-the-Art of Fully Dynamic k-Center

In this section we present the work of Bateni et al. [4], which is the current state of the art for the fully dynamic k-center problem, lowering the amortized update time from the $\Omega\left(n \cdot k^2 \frac{\log \Delta}{\varepsilon}\right)$ of [3] to $O\big(n \cdot k \frac{\log \Delta}{\varepsilon} \log n\big)$ while retaining the same $(2+\varepsilon)$-approximation.

As in the previous section, for ease of readability we will omit the subscript t indicating the time at which each object exists whenever it is not strictly necessary, keeping in mind that, this being a dynamic algorithm, all the presented data structures change over time. We also omit both correctness and running time proofs: the correctness follows from Propositions 2.1 and 2.2, while the running time proofs can be found in the original paper.

The key idea is to abandon the explicit clustering maintained by [3] in favour of the classical reduction from k-center to Maximal Independent Set (MIS) due to Hochbaum and Shmoys [5]. For a guess $\gamma \in \Gamma$ we define the *threshold graph* $G^\gamma = (X, E^\gamma)$ on the active points, where

$$(x, y) \in E^\gamma \iff d(x, y) \leq 2\gamma. \tag{2.2}$$

An independent set of G^γ is then exactly a set of points at pairwise distance $> 2\gamma$, while a *maximal* independent set I^γ enjoys the additional property that every point of X is within distance 2γ of some vertex of I^γ. Therefore, if $|I^\gamma| \leq k$, taking I^γ as centers yields a clustering of radius 2γ; if instead $|I^\gamma| \geq k + 1$, then I^γ contains $k + 1$ points at pairwise distance $> 2\gamma$ and, by Proposition 2.1, $r_k^*(X) > \gamma$. This is precisely the dichotomy required by Proposition 2.2, so maintaining a maximal independent set of G^γ for each $\gamma \in \Gamma$ and returning the centers of the smallest feasible guess gives a $(2 + \varepsilon)$-approximation.

Notice that for the purpose of the dichotomy above it is never necessary to compute the whole MIS: as soon as an independent set of size $k+1$ is found we can already certify $r_k^*(X) > \gamma$. This motivates the *k-Bounded MIS* problem: maintain either a maximal independent set of size at most k, or an arbitrary independent set of size $k + 1$. The advantage is that, while the insertion of a point can create up to $\Omega(n)$ edges in G^γ, only the $O(k)$ potential edges between the new point and the current independent set need to be inspected to decide whether the latter is still maximal.

Maintaining the LFMIS The structure that the algorithm keeps is the *Lexicographically First Maximal Independent Set* (LFMIS) of G^γ with respect to a random ranking $\pi : X \to [0,1]$. The LFMIS is obtained greedily by repeatedly adding the vertex of smallest rank, removing it together with all its neighbours, and iterating until no vertex is left. Each point draws its rank $\pi(x)$ once, uniformly and independently, upon insertion. Maintaining the LFMIS under a random ranking has two convenient properties: it is *history independent*, in that once π is fixed the current LFMIS depends only on the current

graph and not on the order of the updates that produced it; and the insertion of a vertex v is unlikely to cause many changes, since this requires $\pi(v)$ to be small.

In line with the k-Bounded MIS problem, the algorithm only maintains the *top-k LFMIS* L^γ, defined as the first $\min\{k+1, |\mathrm{LFMIS}^\gamma|\}$ vertices of the LFMIS in rank order. The difficulty is to keep track of the "excess" vertices that belong to the LFMIS but not to L^γ, so that they can quickly re-enter L^γ once a vertex of smaller rank leaves it. To this end, for each vertex u not in the LFMIS we define its *eliminator* $\mathrm{elim}^\gamma(u)$ as the neighbour of smallest rank in $N^\gamma(u) \cap \mathrm{LFMIS}^\gamma$, where $N^\gamma(u) = \{x \in X : x \neq u,\ d(u,x) \leq 2\gamma\}$ denotes the neighbourhood of u in G^γ. The eliminator certifies that u cannot enter the independent set. The algorithm maintains a *leader* mapping $\mathrm{lead}^\gamma(u) = \mathrm{elim}^\gamma(u)$, so that whenever a vertex v leaves the LFMIS only the vertices in its *follower set* $F_v^\gamma = \{u : \mathrm{lead}^\gamma(u) = v\}$ need to be examined.

Concretely, the data structure kept for each $\gamma \in \Gamma$ is composed by:

- L^γ, an ordered list of at most $k+1$ vertices, sorted by rank π, holding the top-k LFMIS;
- Q^γ, a priority queue, ordered by rank π, of the unclustered vertices, i.e. the excess vertices waiting to be (re)inserted;
- a follower set F_v^γ for each leader v, together with the leader mapping lead^γ pointing each follower to its eliminator.

Accordingly, at any time each vertex is either an *active leader* (in L^γ), an *inactive leader* (a vertex that still owns a non-empty follower set but has been pushed into Q^γ), a *follower* (a vertex u with $\mathrm{lead}^\gamma(u) \in L^\gamma$), or *unclustered* (in Q^γ with no leader).

Update procedure Every insertion or deletion is handled, for each $\gamma \in \Gamma$, by the procedure ProcessUpdate(v,σ) (Algorithm 6), where $\sigma \in \{+,-\}$ marks the update as an insertion or a deletion. After dispatching the update to Insert or Delete, a final loop drains the queue Q^γ, reinserting its vertices in increasing rank order until either Q^γ is empty or no queued vertex has rank small enough to enter L^γ; this restores the invariant $L^\gamma = \mathrm{LFMIS}^\gamma_{k+1}$.

Algorithm 6: ProcessUpdate(v, σ, γ)

if $\sigma = +$ **then**
  Draw $\pi(v)$ uniformly at random in [0,1]
  $\mathrm{lead}^{\gamma}(v) \leftarrow \perp$
  Insert(v, γ)
else Delete(v, γ);
while $Q^{\gamma} \neq \emptyset$ and $\big(|L^{\gamma}| \leq k$ or $\min_{w \in Q^{\gamma}} \pi(w) < \max_{w \in L^{\gamma}} \pi(w)\big)$ **do**
  $u \leftarrow \arg\min_{w \in Q^{\gamma}} \pi(w)$
  Remove u from Q^{γ}
  Insert(u, γ)

The insertion of a vertex v (Algorithm 7) first discards the trivial case in which L^{γ} is already full and v has the largest rank, so that v cannot belong to the top-k LFMIS and is simply queued. Otherwise it computes the set $S = L^{\gamma} \cap N^{\gamma}(v)$ of the current leaders adjacent to v, which costs $O(k)$ distance queries since $|L^{\gamma}| \leq k + 1$. If $S = \emptyset$ then v has no neighbour in L^{γ} and can enter it, possibly evicting the leader of largest rank into Q^{γ} should L^{γ} exceed $k + 1$ vertices. If instead v has a neighbour of smaller rank, the smallest-rank such neighbour u^* becomes the leader of v. Finally, if v has smaller rank than all of its neighbours in S, it enters the LFMIS and displaces them: the displaced leaders become followers of v, while their own followers, having lost their eliminator, are released into Q^{γ}.

Algorithm 7: INSERT(v, γ)

if $|L^\gamma| = k+1$ and $\pi(v) > \max_{u \in L^\gamma} \pi(u)$ **then**
  Insert v in Q^γ
  return
$S \leftarrow L^\gamma \cap N^\gamma(v)$
if $S = \emptyset$ **then**
  Insert v in L^γ
  if $|L^\gamma| = k+2$ **then**
    $u \leftarrow \arg\max_{u' \in L^\gamma} \pi(u')$
    Remove u from L^γ and insert it in Q^γ
else
  $u^* \leftarrow \arg\min_{u' \in S} \pi(u')$
  if $\pi(u^*) < \pi(v)$ **then**
    if v is a leader **then**
      for $w \in F_v^\gamma$ **do**
        Insert w in Q^γ; $\text{lead}^\gamma(w) \leftarrow \perp$
      Delete F_v^γ
    Add v to $F_{u^*}^\gamma$; $\text{lead}^\gamma(v) \leftarrow u^*$
  else
    for $w \in \cup_{u \in S} F_u^\gamma$ **do**
      Insert w in Q^γ; $\text{lead}^\gamma(w) \leftarrow \perp$
    for $u \in S$ **do**
      $\text{lead}^\gamma(u) \leftarrow v$; remove u from L^γ; delete F_u^γ
    Insert v in L^γ with $F_v^\gamma \leftarrow S$

The deletion of a vertex v (Algorithm 8) is cheap unless v is a leader. If v is a follower it is simply removed from its leader's follower set. If v sits in Q^γ as an inactive leader, its followers are released into Q^γ before v is removed. Finally, if v is an active leader, all of its followers are released into Q^γ and v is removed from L^γ; the queue draining at the end of PROCESSUPDATE then restores the invariant.

Algorithm 8: DELETE(v, γ)

if v is a follower **then**
  Remove v from $F^{\gamma}_{\text{lead}^{\gamma}(v)}$ and from X
  return
else if $v \in Q^{\gamma}$ **then**
  if v is an inactive leader **then**
    for $w \in F^{\gamma}_{v}$ **do**
      Insert w in Q^{γ}; $\text{lead}^{\gamma}(w) \leftarrow \perp$
    Delete F^{γ}_{v}
  Remove v from Q^{γ} and from X
  return
else
  for $w \in F^{\gamma}_{v}$ **do**
    Insert w in Q^{γ}; $\text{lead}^{\gamma}(w) \leftarrow \perp$
  Delete F^{γ}_{v}
  Remove v from L^{γ} and from X

A query is answered exactly as prescribed by Proposition 2.2: we return the centers of the smallest guess $\gamma^* \in \Gamma$ for which $|L^{\gamma^*}| \leq k$, each non-center point x being assigned to the cluster of its leader $\text{lead}^{\gamma^*}(x)$.

The improvement over [3] stems from the random ranking: the cost of an update is, up to an $O(k)$ factor, the number of vertices whose leader changes during that update, and the history independence of the LFMIS guarantees that this number is small in expectation.

Theorem 2.2. *The algorithm described above obtains a $(2+\varepsilon)$-approximation of the k-center Problem and runs in $O\big(n \cdot k \frac{\log \Delta}{\varepsilon} \log n\big)$ amortized time in expectation.*

2.3 With lifetimes

In this section we present our $(2+\varepsilon)$-approximation algorithm for dynamic k-center when viewed in the lifetimes setting, which attains a $(2+\varepsilon)$ approximation ratio in $O((\log \Delta/\varepsilon) \cdot k + \max_t \log |X_t|)$ amortized update time. We recall that in the dynamic with lifetimes setting the algorithm knows $t_{\text{del}}(x) \forall x \in X$ since the point is inserted.

The algorithm is based on the one by Chan et al.[3], presented in section 2.1. In their work, they used the randomness of the reclustering procedure to prove that the points to

be deleted are centers with low enough probability. However, knowing the deletion time of each point may allow us to choose better new centers at reclustering: we can indeed choose the ones with higher deletion times. An effortless reworking of the algorithm, in which the only change is that at line 5 of RandReclust we pick the point with highest deletion time in U, does not achieve $\widetilde{O}(n \cdot k)$ amortized update time. We will show this with a counterexample.

Proposition 2.3. *There exists a sequence of updates on which the modified algorithm runs in $\Theta(n \cdot k^2)$ amortized time.*

Proof. We will give a counterexample that doesn't even take into account the fact that deletion times are known and works with any policy of center selection during reclustering. Let $\gamma \in \mathbb{R}^+$ and $k \in \mathbb{N}^*$, we take $n = 2k$ points $x_1, \ldots, x_n$ s.t.:

- $d(x_{2i-1}, x_{2i}) \leq 2\gamma \ \forall 1 \leq i \leq k$.
- $d(x_i, x_j) > 2\gamma$ for all other pairs of different indexes.
- $t_{\text{arr}}(x_i) = i \ \forall 1 \leq i \leq n$.
- $t_{\text{del}}(x_{2i-1}) = n + k - i + 1 \ \forall 1 \leq i \leq k$.
- $t_{\text{del}}(x_{2i}) = n + 2k - i + 1 \ \forall 1 \leq i \leq k$.

Meaning that the n points are grouped into k clusters of 2 points each. By how the insertion algorithm 3 works, points with index $2i - 1$ will become the i-th center and points with index $2i$ will enter the i-th cluster. Given that the deletions are made in the opposite order of insertion, we will first delete the k-th center, then the $k - 1$-th and so on. The reclustering algorithm 5 only affects clusters with index $\geq$ than the one of the center being deleted, so we are sure that all original centers will remain centers until their deletion. When deleting the i-th center, the only points remaining in clusters with index $j \geq i$ are x_{2j} and they all need to be put on different clusters. So the reclustering of points when deleting the i-th center takes $\Theta((k - i)^2)$ time. The total reclustering cost is then:

$$\sum_{i=1}^{k} \Theta\left((k-i)^2\right) = \Theta\left(k^3\right) = \Theta\left(n \cdot k^2\right)$$

Given that $n = 2k$. □

To fully exploit the deletion time knowledge we will need to modify the deletion algorithm, as well as being more conservative of the times that we call the reclustering procedure.

The data structure maintained for each $\gamma \in \Gamma$ is identical of the one used in [3], the real change is in how it is modified by deletions. To recall we save the sets:

- $C^\gamma = (c_1^\gamma, \ldots, c_{\ell^\gamma}^\gamma)$, a list of $\ell^\gamma \leq k$ centers, such that for each $1 \leq i \neq j \leq \ell^\gamma$, we have $d(c_i^\gamma, c_j^\gamma) > 2\gamma$;
- $D^\gamma = (D_1^\gamma, \ldots, D_{\ell^\gamma}^\gamma)$, a list of $\ell^\gamma \leq k$ disjoint sets of points, such that for each $1 \leq i \leq \ell^\gamma$ and $x \in D_i^\gamma$, we have $d(x, c_i^\gamma) \leq 2\gamma$;
- U^γ, a set of unclustered points such that, for each $x \in U^\gamma$ and $1 \leq i \leq \ell^\gamma$, we have $d(x, c_i^\gamma) > 2\gamma$. Moreover $U^\gamma \neq \emptyset$ implies $\ell^\gamma = k$.

Given that the deletion times are given during insertion the streaming model is a bit different than the previously described ones. Indeed, whilst in the fully dynamic model both insertion and deletions are issued by the model itself, in the setting with lifetimes the deletions need to be handled automatically by the algorithm. To account for that, we save the inserted points in a priority queue, ordered by smallest deletion time, which is the reason we get an additive $O(\max_t \log |X_t|)$ in the final complexity.

Update procedure Procedure $\textsc{Update}_{2+\varepsilon}(p,t)$ (see Algorithm 9 for the pseudocode) is called when a new point p arrives (in this case $t = t_{\text{arr}}(p)$), or at any time t when expired points, if any, need to be deleted from the data structure (in this case, $p = \textsc{null}$). First, for each $q \in Q$ with $t_{\text{del}}(q) \leq t$, q is removed from Q and procedure $\textsc{Delete}(q, \gamma)$ is invoked for each $\gamma \in \Gamma$. Then, if $p \neq \textsc{null}$, procedure $\textsc{Insert}(p, \gamma)$ is invoked for each $\gamma \in \Gamma$, and p is added to the priority queue Q.

$\textsc{Insert}(p,\gamma)$ (see Algorithm 10 for the pseudocode) inserts p in the cluster of D^γ of lowest index whose center is at distance at most 2γ from p. If no such cluster exists, then if D^γ comprises $\ell^\gamma < k$ clusters, then the procedure creates a new cluster by making p its center, otherwise, it inserts p in U^γ. At the end of $\textsc{Insert}(p,\gamma)$ a reclustering procedure, described later, is called.

$\textsc{Delete}(p,\gamma)$, where p is an expired point, works as follows (see Algorithm 11). If p is not a center, it is simply removed from its cluster in D^γ. If p is a center c_i^γ, the procedure deletes c_i^γ and tries to recluster all points of D_i^γ in clusters with index $> i$. Intuitively, each of these points is inserted in the cluster where it would have been inserted if c_i^γ had

Algorithm 9: UPDATE$_{2+\varepsilon}(p,t)$

```
1 while min_{q∈Q} t_del(q) ≤ t do
2 |   Remove q from Q
3 |   for γ ∈ Γ do
4 |   |   DELETE(q, γ)
5 if p ≠ NULL then
6 |   for γ ∈ Γ do
7 |   |   INSERT(p, γ)
8 |   Insert p in Q
```

Algorithm 10: INSERT(p,γ)

```
1 for i = 1, ... ℓ^γ do
2 |   if d(p, c_i^γ) ≤ 2γ then
3 |   |   Insert p in D_i^γ, and goto
  |   |     line 8
4 if ℓ^γ < k then
5 |   Insert p in C^γ and in D^γ_{ℓ^γ+1}
6 |   ℓ^γ ← ℓ^γ + 1
7 else Insert p in U^γ;
8 RECLUSTERING(γ)
```

not existed. Then, all cluster indexes greater than i are decremented by 1, making them contiguous again, and, if $U^\gamma \neq \emptyset$, the point of U^γ with the latest deletion time becomes the center of a new cluster. Finally, the reclustering procedure is called.

Note that the expensive part of the deletion procedure, namely, re-assigning points and creating a new cluster, only happens if a cluster center is deleted. In order to limit these costly operations, we aim at selecting centers with largest deletion times. In particular, this is enforced when a new center from U^γ must be selected (Lines 17÷23 of DELETE(p,γ)), and when the reclustering is performed at the end of INSERT and DELETE, which is described next.

Definition 2.1. *For a given $\gamma \in \Gamma$, we define as **persistent** those clustered points which expire after their respective clusters' centers, and **vanishing** those which either expire earlier than their clusters' centers or are unclustered.*

Note that only persistent points can change cluster because of the deletion of their center, as vanishing points expire before the deletion of their center. We partition each cluster D_i^γ into persistent and vanishing points as $P_i^\gamma := \{x \in D_i^\gamma :\ t_{\text{del}}(c_i^\gamma) < t_{\text{del}}(x)\}$ and $V_i^\gamma := \{x \in D_i^\gamma :\ t_{\text{del}}(c_i^\gamma) \geq t_{\text{del}}(x)\}$. In practice, we maintain two counters per cluster for $|P_i^\gamma|$ and $|V_i^\gamma|$, which can be updated in $O(1)$ time for each insertion and deletion.

Procedure RECLUSTERING(γ) (see Algorithm 12), first searches for the smallest index i such that

$$\sum_{j=i}^{\ell^\gamma} |P_j^\gamma| > |U^\gamma| + \sum_{j=i}^{\ell^\gamma} |V_j^\gamma|. \tag{2.3}$$

If such i exists, then it reclusters the points of $U^\gamma \cup (\cup_{j=i}^{\ell^\gamma} D_j^\gamma)$ into $\ell^\gamma - i + 1$ clusters, selecting, for each such new cluster, a center with latest deletion time. Note that, after reclustering, some points might remain unclustered, but the newly created clusters comprise only vanishing points.

Algorithm 11: DELETE(p, γ)

1 **if** $p \notin C^\gamma$ **then**
2 Remove p from U^γ or D^γ
3 **goto** line 24
4 Let i be such that $p = c_i^\gamma$
5 **for** $x \in D_i^\gamma$ with $x \neq p$ **do**
6 **for** $j = i+1, \ldots, \ell^\gamma$ **do**
7 **if** $d(x,c_j^\gamma) \leq 2\gamma$ **then**
8 Insert x in D_j^γ, and **break**
9 **if** *x was not inserted* **then**
10 **if** $\ell^\gamma < k$ **then**
11 Insert x in C^γ and in $D_{\ell^\gamma+1}^\gamma$;
12 $\ell^\gamma \leftarrow \ell^\gamma + 1$
13 **else** Insert x in U^γ;
14 Remove c_i^γ and D_i^γ;
15 $\ell^\gamma \leftarrow \ell^\gamma - 1$;
16 Rename indexes of C^γ, D^γ to be contiguous
17 **if** $U^\gamma \neq \emptyset$ **then**
18 $\ell^\gamma \leftarrow \ell^\gamma + 1$;
19 Pick $u \in U^\gamma$ s.t. $t_{\text{del}}(u)$ is maximal;
20 Remove u from U^γ, insert in C^γ, D^γ as $c_{\ell^\gamma}^\gamma$
21 **for** $x \in U^\gamma$ **do**
22 **if** $d(x,u) \leq 2\gamma$ **then**
23 Remove x from U^γ, insert in $D_{\ell^\gamma}^\gamma$
24 RECLUSTERING(γ)

Algorithm 12: RECLUSTERING(γ)

1 $i \leftarrow$ min index h s.t. $\sum_{j=h}^{\ell^\gamma} |P_j^\gamma| > |U^\gamma| + \sum_{j=h}^{\ell^\gamma} |V_j^\gamma|$
2 **if** *no such i exists* **then**
3 **return**
4 $U' \leftarrow U^\gamma \cup (\cup_{j=i}^{\ell^\gamma} D_j^\gamma)$
5 Remove centers c_j^γ and clusters D_j^γ $\forall j \geq i$;
6 $\ell^\gamma \leftarrow i - 1$;
7 **for** $j = i, \ldots, k$ **do**
8 **if** $U' = \emptyset$ **then**
9 **break**
10 $\ell^\gamma \leftarrow j$;
11 Pick $u \in U'$ s.t. $t_{\text{del}}(u)$ is maximal;
12 Remove u from U', insert in C^γ, D^γ as c_j^γ;
13 **for** $x \in U'$ **do**
14 **if** $d(x,u) \leq 2\gamma$ **then**
15 Remove x from U', insert in D_j^γ
16 $U^\gamma \leftarrow U'$

Algorithm 13: QUERY$_{2+\varepsilon}(t)$

1 $\gamma^* \leftarrow \min\{\gamma \in \Gamma \ : \ U^\gamma = \emptyset\}$;
2 **return** C^{γ^*}

The following remark shows that the reclustering is crucial to obtain sublinear amortized update time.

Remark. *There exists a sequence of $O(n)$ calls to* UPDATE$_{2+\varepsilon}$ *which, without the use of* RECLUSTERING*, would perform a total of $\Theta(n^2)$ operations.*

Proof. We describe an adversarial point set for a given guess $\gamma \in \Gamma$. At time $t = 1$, we insert a point p_1 with lifetime such that $t_{\text{del}}(p_1) = t_{\text{arr}}(p_1) + n + 1$.

At time $t = 2, \ldots, n$, we insert $n - 1$ points $p_2, \ldots, p_n$, each with lifetime such that $t_{\text{del}}(p) = t_{\text{arr}}(p) + 2n - 1$, and such that each p_i with $i > 1$ is at distance $3/2\gamma$ from p_1 (and, e.g., at distances $\gamma/10$ between each other). Clearly, all these points are assigned to cluster D_1, with the first point p_1 being the center. Then, at time $t = n+1, \ldots, 2n$, we

insert n additional points $p_{n+1}, \ldots, p_{2n}$, each with lifetime such that $t_{\text{del}}(p_i) = t_{\text{arr}}(p_i) + 2$ and such that each such p_i is at distance $5/2\gamma$ from p_1, at distance $5/2\gamma$ from the other points p_j with $n < j \leq 2n$, and at distance $3/2\gamma$ from points $p_2, \ldots, p_n$.

The first n calls of $\text{UPDATE}_{2+\varepsilon}$ involve no deletions, so they take $O(k)$ time each. The call at time $n+1$ inserts p_{n+1} as a new center, since it is at distance $> 2\gamma$ from p_1, in $O(k)$ time. However, the call at time $n+2$ first involves the deletion of p_1 and moving $p_2, \ldots, p_n$ to the cluster with center p_{n+1}, and then inserts p_{n+2} as a new center. This takes $\Theta(n)$ time. Each call at time $t = n+3, \ldots, 2n$ involves the deletion of point p_{t-2}, moving $p_2, \ldots, p_n$ to the cluster with center p_{t-1} and the insertion of p_t as a new center, which takes $\Theta(n)$. Therefore, the algorithm performs a total of $\Theta(n^2)$ operations. □

Query procedure At any time t, an approximate solution to k-center for the active set X_t can be computed calling $\text{QUERY}_{2+\varepsilon}(t)$ (see Algorithm 13 for the pseudocode). We assume that when $\text{QUERY}_{2+\varepsilon}(t)$ is invoked, the data structure contains no expired points. This can be ensured by checking that the minimum deletion time of a point in Q is $> t$, and, if this is not the case, invoking $\text{UPDATE}_{2+\varepsilon}(\text{NULL},t)$ before $\text{QUERY}_{2+\varepsilon}(t)$. The procedure simply returns the set C^{γ^*}, with $\gamma^* := \min_{\gamma \in \Gamma}\{\gamma\colon\ U^\gamma = \emptyset\}$, as a set of centers.

The following subsections analyze the approximation ratio, and the time and memory requirements of the above algorithm. For convenience, we define a *time step* as one call of INSERT or DELETE. Since arrival times are distinct and Q breaks ties for deletion times based on arrival times, we can assume, without loss of generality, that each call corresponds to a unique time step. For any time step $t \geq 0$, we use X_t to denote the active set, and $C^{\gamma,t}, D^{\gamma,t}, U^{\gamma,t}, Q^t$ to denote the various components of the data structure, *after* the t_{th} call of INSERT or DELETE. Time step $t = 0$ refers to the initial phase of the algorithm, when nothing is yet inserted. We may drop t when clear from the context.

2.3.1 Approximation

First, we prove that the algorithm maintains the following invariant. This can be then used to prove the approximation guarantee for $\text{QUERY}_{2+\varepsilon}$.

Lemma 2.1. *For any time step t, the following properties hold for each $\gamma \in \Gamma$, and $x \in X_t$:*

1. *if $d(x, C^{\gamma,t}) > 2\gamma$, then $x \in U^{\gamma,t}$;*
2. *if $d(x, C^{\gamma,t}) \leq 2\gamma$, then $x \in D_i^{\gamma,t}$ with $i = \arg\min_{j=1,\ldots\ell^{\gamma,t}}\{c_j \in C^{\gamma,t}\colon\ d(x, c_j) \leq 2\gamma\}$.*

Moreover, if $U^{\gamma,t} \neq \emptyset$ *then* $|C^{\gamma,t}| = k$.

Proof. At step $t = 0$, the property holds by vacuity. Suppose inductively that at step $t \geq 0$, the invariant holds for all points in X_t. We recall that the superscript (e.g. $U^{\gamma,t}$) denotes the state of a set *after* the t-th insert or delete operation. When the superscript is missing, it denotes a state during the execution of the operations. At time step $t + 1$:

- If INSERT(p, γ) is called, then all points in X_t are left untouched. Moreover, the invariant for p holds by construction.
- If DELETE(p, γ) is called and p is not a center, then all points in $X_t \setminus \{p\}$ are left untouched. Let instead p be a center $c_i^{\gamma,t}$. Then, for each $x \notin D_i^{\gamma,t} \cup U^{\gamma,t}$, note that the center and cluster index renaming does not affect the invariant. For each $x \in D_i^{\gamma,t}$, either x is inserted in D_h with $h = \arg\min_{j=i+1,\dots\ell}\{c_j \in C\colon\ d(x, c_j) \leq 2\gamma\}$ (possibly by creating new clusters), or it is inserted in U if no center at distance $\leq 2\gamma$ is found. In the first case, note that $\min_{j=1,\dots,i-1} d(x, c_j) > 2\gamma$, otherwise by the inductive hypothesis x would not belong to D_i. Therefore, we have that $h = \arg\min_{j=1,\dots\ell}\{c_j \in C\colon\ d(x, c_j) \leq 2\gamma\}$, thus fulfilling the invariant.

  Finally, let $U \neq \emptyset$ and consider a $x \in U$, either because it belonged in $U^{\gamma,t}$ or because it was inserted in U on line 13. In this case, one point u will be selected as a new center, and thus $|C| = k$. We have that if $d(x, C^{\gamma,t+1}) > 2\gamma$, then we have $d(x, u) > 2\gamma$ and therefore x remains in U, satisfying the first case of the invariant. Otherwise, since $x \in U$ implies that $d(x, c_i) > 2\gamma\ \forall i = 1, \dots, \ell - 1$, we have that $d(x, u) \leq 2\gamma$, and thus x is inserted in D_ℓ, thus satisfying the second case of the invariant.

Finally, after a call to INSERT or DELETE, the RECLUSTERING(γ) procedure might be called. In that case, we note that each $x \in \bigcup_{j=1}^{i-1} D_j^{\gamma,t}$ is left untouched, and the invariant is still valid. On the other hand, if $x \in U'$ after line 4, then it could not be inserted in any of the clusters D_j for $j = 1, \dots, i-1$, by the inductive hypothesis. Moreover, by construction it will be inserted in the cluster D_h with $h = \arg\min_{j=i,\dots\ell}\{c_j \in C\colon\ d(x, c_j) \leq 2\gamma\}$, as a point can be inserted in a cluster only if it is not inserted in a cluster with a lower index. At the end of the loop (i.e., $|C| = k$), any point x such that $d(x, C) > 2\gamma$ is left in U.

We therefore have the claim by induction. □

Theorem 2.3. *For any time step* t*, the solution returned by* QUERY$_{2+\varepsilon}(t)$ *is a* $(2 + \varepsilon)$*-approximation for the* k*-center problem on the active set* X_t.

Proof. Recall that the algorithm returns $C^{\gamma^*,t}$, with $\gamma^* = \min_{\gamma\in\Gamma}\{\gamma\colon\ U^{\gamma,t} = \emptyset\}$. If $\gamma^* = d_{\min}$, the theorem trivially holds. Suppose $\gamma^* > d_{\min}$, and let $\gamma' = \gamma^*/(1+\beta) \in \Gamma$. Clearly, $U^{\gamma',t} \neq \emptyset$, hence $|C^{\gamma',t}| = k$. Let $W = C^{\gamma',t} \cup \{x\}$, for an arbitrary $x \in U^{\gamma',t}$. By construction and by Lemma 2.1 we have that $\forall p \neq q \in W$, $d(p,q) > 2\gamma'$. Moreover, since $|W| = k+1$, we have that $r_k^*(X_t) \geq \min_{p\neq q\in W} d(p,q)/2 > \gamma' = \gamma^*/(1+\beta)$, as at least two distinct points of W are in the same optimal cluster. By Lemma 2.1, $\max_{x\in X_t} d(x, C^{\gamma^*,t}) \leq 2\gamma^* < 2(1+\beta)r_k^*(X_t) \leq (2+\varepsilon)r_k^*(X_t)$. □

2.3.2 Running time and memory complexity

Consider a sequence of n consecutive calls to the UPDATE$_{2+\varepsilon}$ procedure, which must involve $\Theta(n)$ insertions and $O(n)$ deletions. In what follows, we analyze the performance of our algorithm relatively to these calls. Let us focus on an arbitrary guess $\gamma \in \Gamma$. We first analyze the overall cost of the reclustering operations.

Lemma 2.2. *After a call of* RECLUSTERING(γ) *every point that was reclustered, i.e., every point belonging to the set U' in $\bigcup_{j=i}^{\ell^\gamma} D_j^\gamma \ \cup U^\gamma$ before the reclustering, becomes vanishing for γ.*

Proof. Since at each step j the new center c_j gets chosen from all remaining points as the one with maximal deletion time, any point x that gets inserted into D_j must have deletion time $t_{\text{del}}(x) < t_{\text{del}}(c_j)$, and is therefore vanishing. If a point is left in U, then it is vanishing by definition. □

Lemma 2.3. *If a point $x \in X$ is vanishing for γ at time t_0, then x remains vanishing for all t such that $t_0 \leq t < t_{\text{del}}(x)$.*

Proof. We will show that the vanishing property (for a fixed guess γ) is invariant under the call of INSERT, DELETE or RECLUSTERING algorithms, unless $t = t_{\text{del}}(x)$ and the point is thus removed. The property holds at $t = 0$ by vacuity and the proof follows by induction.

Fix some $x \in X$ which is vanishing at time t.

- **Insertion**: If x is vanishing then it is already inserted, so any insertion operation will regard a new point and cannot modify either x or its center. Then, x will remain vanishing after any insertion operation.

- **Deletion**: Suppose that we call DELETE(p, γ). If $p = x$ the current time is $t_{\text{del}}(x)$ and the proposition holds. Furthermore, if $p \notin C^\gamma$ (i.e. p is not a center) its deletion

will only alter p itself and no other point. Therefore, we can assume $p \neq x$ and $\exists i$ s.t. $p = c_i$. We then have 3 cases:

1. $x \in D_i$. This would mean that the center of x has a deletion time smaller than that of x, which is in contradiction to the fact that x is vanishing.
2. $x \in D_j$, for some $j \neq i$. In the case that $p = c_i$ the deletion algorithm affects clusters with index different from i by possibly adding more points to them (Line 8) and by relabeling their indexes (Line 16). Therefore, no changes are made to x or to its center c_j, and thus x remains vanishing.
3. $x \in U$. In this case, x remains in U or is inserted in D_l (Line 23). Either way, it remains vanishing, since we picked $c_l = u$ s.t. $t_{\text{del}}(u)$ is maximal among all points in U.

- **Reclustering**: Suppose that we call RECLUSTERING(i,γ). If $\exists j < i$ such that $x \in D_j$, then the reclustering operation does not affect D_j in any way, meaning that x remains vanishing. The other case is covered by Lemma 2.2, and thus x remains vanishing.

□

Proposition 2.4. *The total number of operations made by all calls of* RECLUSTERING(γ) *is* $O(n \cdot k)$.

Proof. The main for loop of Algorithm 12 is repeated $O(k)$ times. In each iteration we need to scan U' linearly to first find the element with maximal t_{del} and then to check whether some other point in U' is close to it. So the main loop is executed in $O(k \cdot |U'|)$. The other parts of the algorithm can be executed in $O(|U'|)$, for a total running time of $O(k \cdot |U'|)$. Then:

$$k|U'| = k\left|U \cup \bigcup_{j=i}^{k} D_j\right| \leq k\left(|U| + \sum_{j=i}^{k}(|P_j| + |V_j|)\right) \leq 2k\sum_{j=i}^{k}|P_j|,$$

where the last inequality is guaranteed by the fact that Property (2.3) holds for i when calling the reclustering.

This means that the number of operations is $O\left(k \cdot \sum_{j=i}^{l}|P_j|\right)$ and thus bounded by the number of persistent points involved in the reclustering. Given that by Lemma 2.2 all persistent points in $P_{j\geq i}$ will become vanishing and that by Lemma 2.3 every vanishing point remains vanishing, the number of persistent points in all reclusterings is bounded

by n and thus the total number of operations made by all calls of RECLUSTERING(i,γ) is $O(n \cdot k)$ □

We now estimate the overall costs of insertions and deletions. For insertions, the following proposition holds because of Proposition 2.4 and because, besides the call to RECLUSTERING(γ), INSERT(p,γ) takes $O(k)$ time.

Proposition 2.5. *The total number of operations made by all calls of* INSERT(p,γ) *is* $O(n \cdot k)$.

Proof. The proof is trivial since the insertion algorithm simply scans the up to k centers present in the data structure, which is $O(k)$. It can also insert the point inside a cluster or create a new cluster, both of which are $O(1)$ operations. Given that the total number of insertions is n, the total number of operations made by all calls of INSERT(p,γ) must be $O(n \cdot k)$. □

Consider now the deletions. In what follows, for time t and $x \in X_t$, and for each $\gamma \in \Gamma$, let $c_{ID}^{\gamma,t}(x)$ be the index of the cluster in D^γ that x is in at time t, and let $c_{ID}^{\gamma,t}(x) = k+1$ if $x \in U^\gamma$.

Lemma 2.4. *Let $x \in X$ and $\gamma \in \Gamma$. Call $\mathcal{Q}_x^\gamma$ the set of centers c such that at time $t = t_{\text{del}}(c) - 1$ it holds that x is vanishing for γ and $c_{ID}^{\gamma,t}(c) \leq c_{ID}^{\gamma,t}(x)$. Then, $|\mathcal{Q}_x^\gamma| \leq k+1$.*

Proof. We will show that for $c \neq x$ to satisfy the two properties implies the fact that when c was last picked as a center x was either not inserted or persistent. Suppose by contradiction that, when c was last picked as a center, x was vanishing. We have 3 possibilities:

1. c became a center for the last time at time t' during an insertion operation (Line 5) or during a re-assignment of persistent points after a deletion (Line 12). In both of these cases $c_{ID}^{\gamma,t'}(c) > c_{ID}^{\gamma,t'}(x)$ as c becomes the center of the cluster with the highest index. Given that we are assuming that t' is the last time that c was chosen as a center, there cannot be any reclustering involving c after time t'. Furthermore, if no reclustering is made, clusters stay in their relative order and point x will remain inside its cluster until its deletion. So the fact that $c_{ID}^{\gamma,t'}(c) \geq c_{ID}^{\gamma,t'}(x)$ is in contradiction with $c_{ID}^{\gamma,t}(c) \leq c_{ID}^{\gamma,t}(x)$.

2. c became a center for the last time after a deletion. If $x \notin U$ the analysis is akin to the one in the first case. If $x \in U$ we know that $t_{\text{del}}(x) < t_{\text{del}}(c)$, thus at time t x will already be deleted and therefore cannot be vanishing.

3. c became a center for the last time during the execution of RECLUSTERING(i,γ). If $x \in D_j$ s.t. $j < i$ the analysis is akin to the one in the first case. Let instead $x \in U'$ (same U' as in Algorithm 12). If x ends up in a cluster with smaller index than the one of c, then we break the assumption that $c_{ID}^{\gamma,t}(c) \leq c_{ID}^{\gamma,t}(x)$. If instead it ends up in a cluster with index greater or equal to the one of c, then we must have $t_{\text{del}}(x) < t_{\text{del}}(c)$, and thus x cannot be vanishing at $t = t_{\text{del}}(c) - 1$ because it has been already deleted.

Given that by Lemma 2.3 a vanishing point remains vanishing, calling t' the time at which x becomes vanishing, every center c which satisfies the two properties of the Lemma must be picked at time $< t'$ and carried on up to a time $\geq t'$. Given that at most k centers can coexist simultaneously, the maximum number of centers $c \neq x$ satisfying the properties is k. Accounting for the possibility that $c = x$, we obtain at most $k + 1$ centers. □

Proposition 2.6. *The total number of operations made by all calls of* DELETE(p,γ) *is* $O(n \cdot k)$.

Proof. First, we bound the number of operations done by the re-assignment of cluster points in the loop at Line 5. We begin by noticing that when c_i is deleted, only persistent points contribute to the loop at Line 5. For a single such point $x \in D_i, x \neq c_i$, Lines 6-13 perform $O\big(c_{ID}^{\gamma,t}(x) - c_{ID}^{\gamma,t-1}(x)\big)$ operations, i.e., proportional to the difference between the current index i and the index of the cluster that it will end up in. If $c_{ID}^{\gamma,t}(x)$ was non-decreasing over time, then the maximum number of operations for x across all timesteps would be $O(k)$. However this is not the case because of re-indexing due to center deletions.

Given that x must be a persistent point, we are only interested in events in which $c_{ID}^{\gamma,t}(x)$ decreases and x remains a persistent point. The only such case occurs during the relabeling process in a deletion that involves a center (Line 16) with index $< c_{ID}^{\gamma,t-1}(x)$. In this case $c_{ID}^{\gamma,t}(x) = c_{ID}^{\gamma,t-1}(x) - 1$, so each center deletion adds 1 to the maximum number of re-assignment operations for each persistent point with index greater or equal to its own.

Let $\mathcal{D} := \{c : c \in C^{\gamma,t_{\text{del}}(c)-1}\}$ be the set of all deleted centers at any time. We remark that, for $x \in X$, superscripts $t_{\text{del}}(x) - 1$ denote states right before the deletion of x. We get that the total number of re-assignment operations is:

$$O\Bigg(n \cdot k + \sum_{c \in \mathcal{D}} \sum_{i \geq c_{ID}^{\gamma,t_{\text{del}}(c)-1}(c)} |P_i^{\gamma,t_{\text{del}}(c)-1}|\Bigg) = O\Bigg(nk + \sum_{c \in \mathcal{D}} \Big(|U^{\gamma,t_{\text{del}}(c)-1}| + \sum_{i \geq c_{ID}^{\gamma,t_{\text{del}}(c)-1}(c)} |V_i^{\gamma,t_{\text{del}}(c)-1}|\Big)\Bigg),$$

where we used the Property (2.3). If we fix $c \in \mathcal{D}$ in the last summation and call $j := c_{ID}^{\gamma,t_{\text{del}}(c)-1}(c)$ we get that, for each $x \in U^{\gamma,t_{\text{del}}(c)-1} \cup \bigcup V_{i\geq j}^{\gamma,t_{\text{del}}(c)-1}$, x is by definition vanishing and $c_{ID}^{\gamma,t_{\text{del}}(c)-1}(c) \leq c_{ID}^{\gamma,t_{\text{del}}(c)-1}(x)$. Therefore we have:

$$\sum_{c\in\mathcal{D}}\left(|U^{\gamma,t_{\text{del}}(c)-1}| + \sum_{i\geq c_{ID}^{\gamma,t_{\text{del}}(c)-1}(c)} |V_i^{\gamma,t_{\text{del}}(c)-1}|\right) \leq \sum_{x\in X}\left|\left\{c\in\mathcal{D}: \begin{smallmatrix} x \text{ is vanishing at time } t_{\text{del}}(c)-1 \\ \text{and } c_{ID}^{\gamma,t_{\text{del}}(c)-1}(c)\leq c_{ID}^{\gamma,t_{\text{del}}(c)-1}(x)\end{smallmatrix}\right\}\right|$$
$$= \sum_{x\in X}|\mathcal{Q}_x^\gamma| \leq n\cdot(k+1),$$

where we used a double counting argument by rearranging the summation order, and then applied Lemma 2.4. Hence the total number of operations made by the re-assignment of cluster points in the deleting function over all calls of DELETE(p,γ) is $O(n\cdot k)$. Second, we bound the number of operations made by the creation of a new cluster in the deleting function (Line 17). When creating a new cluster we scan each element in U for finding the one with greater t_{del} and then we do the same for finding which points are within distance 2γ. Therefore, the total number of operations is $O(\sum_{c\in\mathcal{D}}|U^{\gamma,t_{\text{del}}(c)-1}|)$, which, as above, is $O(n\cdot k)$.

Finally, all other operations in the deletion procedure take $O(k)$ time, which concludes the proof. □

Based on Propositions 2.5 and 2.6, we can prove the bound for update times. We also state the bound for query times, as well as the memory requirements of our algorithm.

Theorem 2.4. *The total number of operations made by a sequence of n consecutive calls to* UPDATE$_{2+\varepsilon}$ *is* $O(n(\max_t \log|X_t| + (\log\Delta/\varepsilon)k))$.

Proof. By implementing the priority queue Q as a heap, each of the $\Theta(n)$ insertions and $O(n)$ remove-min operations in Q costs $O(\log|X_t|)$, so their overall cost is $O(n\cdot\max_t\log|X_t|)$. By Propositions 2.5 and 2.6, for each $\gamma\in\Gamma$, we have that the overall costs of all calls INSERT(p,γ) and DELETE(p,γ) is $O(nk)$. Since $|\Gamma| \in O\big(\log_{1+\varepsilon}\Delta\big) = O(\log\Delta/\varepsilon)$, the theorem follows. □

Theorem 2.5. *The number of operations made by each* QUERY$_{2+\varepsilon}$ *call is* $O((\log\Delta/\varepsilon)+k)$.

Proof. Note that we iterate through the guesses $\gamma\in\Gamma$, and for each guess we check in constant time the size of C^γ. This takes $O(|\Gamma|) = O(\log\Delta/\varepsilon)$ time. Finally, returning the solution takes $O(|C^{\gamma^*}|) = O(k)$ time. □

Theorem 2.6. *The maximum number of points maintained in memory when processing the pointset X is $O((\log \Delta/\varepsilon) \max_t |X_t|)$.*

Proof. Note that, by construction, after the call of UPDATE(p, t), only points of X_t are maintained in memory. Moreover, for a fixed guess γ, any such point $p \in X_t$ can belong, by construction, to only one of the sets $D_i^\gamma, i = 1, \ldots, \ell^\gamma$ or U^γ. We therefore have that $\sum_{i=1}^{\ell^\gamma} |D_i^\gamma| + |U^\gamma| \leq |X_t|$. Moreover, clearly $|C^\gamma| \leq |X_t|$ as the centers are all distinct. Since we have $|\Gamma| \in O(\log \Delta/\varepsilon)$, the data structure maintains at most $O((\log \Delta/\varepsilon)|X_t|)$ points. Moreover, the priority queue Q contains each point of the active set exactly once, for a total of $|X_t|$ points. Finally, until the following call of UPDATE, the number of points stored in the data structures can only decrease (i.e., if QUERY calls DELETE), thus concluding the proof. □

Interestingly, our update times are tight, up to a factor $O(\log \Delta/\varepsilon)$, due to the lower bound of $\Omega(|X|k)$ distance queries given by [4, Theorem 54], which holds even in the offline setting.

2.4 Bonus: A new fully dynamic algorithm for k-center

The knowledge of the deletion times in the previously described algorithm is used in 2 cases:

1. When choosing a new center from U we need to find the one with higher deletion time (Line 19 of Algorithm 11 and Line 11 of Algorithm 12).

2. In the definition of persistent and vanishing points (Definition 2.1).

By modifying these 2 parts of our algorithm we can obtain a fully dynamic algorithm that does not need to know deletion times in advance:

1. Points from U can be chosen randomly, just like in the algorithm from Chan et al. (Section 2.1).

2. Vanishing points are defined as the points that entered a cluster after choosing a new center from U (Line 23 of Algorithm 11 and Line 15 of Algorithm 12), persistent points are all the remaining ones.

The rest of the algorithm remains the same.

We conjecture that this variant of our algorithm achieves the same update running time of $O(n(\max_t \log |X_t| + (\log \Delta/\varepsilon)k))$ in expectation, lowering the SOTA running time for fully dynamic algorithms. Sadly, we didn't manage to prove or disprove this conjecture, but we will still include this variant in the experiments to see if it has some practical applications.

2.4.1 Comparison with previous work

Our algorithm mimics the fully-dynamic algorithm from [3], yet by exploiting the known deletion times it lowers the amortized update times from $O\big(\frac{\log \Delta}{\varepsilon} k^2\big)$ to $O\big(\frac{\log \Delta}{\varepsilon} k\big)$, while having the same approximation ratio. In fact, this allows us to have faster updates even compared to the more complex $(2+\varepsilon)$-approximation algorithm by [4], which has amortized $O\big((\log \Delta/\varepsilon) \max_t \log |X_t| (k + \max_t \log |X_t|)\big)$ update time. Importantly, since our algorithm is deterministic, it works even against metric-adaptive adversaries, while the aforementioned two can only deal with oblivious ones. Therefore, in settings where lifetimes are available, it would be wasteful to use a fully-dynamic algorithm, as they are more computationally expensive and with less guarantees, rather than our algorithm with lifetimes. In fact, making use of lifetimes, our algorithm can *sidestep* the metric-adaptive lower bound by [4, Thm. 3], for which no fully-dynamic algorithm with $O(\text{polylog}|X_t|)$ update times can have $O(1)$ approximation for large enough k.

The recent preprint by [6] addresses a dynamic setting akin to the one with lifetimes, while focusing primarily on the memory requirements. Indeed, their algorithm requires $O((\log \Delta/\varepsilon) \cdot k^3)$ memory, as opposed to our requirements, which are linear in the active set size. However, their algorithm features a high $O((\log \Delta/\varepsilon) \cdot k^4)$ update time, and, more importantly, features only a $(6k + 2 + \varepsilon)$ approximation ratio, which is much worse than our almost-tight $(2+\varepsilon)$-approximation.

Chapter 3

Dynamic k-Center with Sublinear Memory

This chapter is dedicated to analyzing memory efficient approaches to the k-Center problem. The algorithms presented in the previous one worked in total memory linear in the number of active points $|X_t|$. However, when dealing with large datasets, a working memory linear in the size of the active set might not fit in the fast access memory. In order to develop algorithm that utilize sublinear working memory a compromise on the approximation accuracy must be done. We will now see two $(6+\varepsilon)$- approximation algorithms that build on each other: the first one by Cohen-Addad et al.[2] is limited to the sliding window setting (a special case of the lifetimes setting), the second one is our extension in the lifetimes setting.

3.1 Sliding Window

In this section we present the work of Cohen-Addad et al.[2], who gave the first algorithm for the k-center problem in the *sliding window* model, achieving a $(6+\varepsilon)$-approximation while storing only $O((k/\varepsilon)\log\Delta)$ points, i.e., a number of points independent of the size of the active set. As in the previous chapter, for ease of readability we will omit the subscript t indicating the time at which each object exists whenever it is not strictly necessary, keeping in mind that, this being a dynamic algorithm, all the presented data structures change over time. We also omit both correctness and running time proofs as they both follow as a special case of the analysis with lifetimes or can be found on the original paper [2]. In the sliding window model, the stream is a (potentially infinite)

sequence of points and, for a fixed integer $N > 0$ called the *window size*, the active set X_t consists of the N most recent points of the stream. Equivalently, every point p satisfies $t_{\mathrm{del}}(p) = t_{\mathrm{arr}}(p) + N$, so that points expire in the very same order in which they arrive (FIFO). In particular, at each time step exactly one point arrives and at most one point expires, and the deletion order is completely determined by the arrival order.

As in the algorithms of the previous chapter, for each guess $\gamma \in \Gamma$, defined by Equation 2.1 and here instantiated with $\beta = \varepsilon/6$, the algorithm maintains a data structure that either exhibits $k+1$ points at pairwise distance $> 2\gamma$, which by Proposition 2.1 certifies $r_k^*(X_t) > \gamma$, or produces a clustering of radius 6γ. Unlike the $(2+\varepsilon)$-approximations seen so far, the second branch does not yield a clustering of radius 2γ, which is precisely the reason for the looser approximation factor.

The data structure kept for each $\gamma \in \Gamma$ is composed by:

- $A^\gamma = (a_1^\gamma, \ldots, a_{\ell^\gamma}^\gamma) \subseteq X_t$, a list of $\ell^\gamma \leq k+1$ *attractors* such that $d(a_i^\gamma, a_j^\gamma) > 2\gamma$ for all $i \neq j$. For $x \in X_t$ and $a \in A^\gamma$, we say that x is *attracted* by a if $t_{\mathrm{arr}}(x) \geq t_{\mathrm{arr}}(a)$ and $d(x,a) \leq 2\gamma$.

- $R_{\mathrm{act}}^\gamma = (r_1^\gamma, \ldots, r_{\ell^\gamma}^\gamma) \subseteq X_t$, a set of *active representatives*, where, for $1 \leq i \leq \ell^\gamma$, $r_i^\gamma = rep^\gamma(a_i^\gamma)$ is the point with latest deletion time (equivalently, the most recent point) among those attracted by a_i^γ.

- $R_{\mathrm{orph}}^\gamma \subseteq X_t$, a set of *orphaned representatives*, i.e., representatives whose attractor has expired but which are themselves still in the window. We define $R^\gamma = R_{\mathrm{act}}^\gamma \cup R_{\mathrm{orph}}^\gamma$.

As before, if $|A^\gamma| \geq k+1$ then the attractors are $k+1$ points at pairwise distance $> 2\gamma$ and form the desired certificate, so we only ever need to produce a clustering when $|A^\gamma| \leq k$. In this case, as we will show in the next section (Lemma 3.1), every active point lies within distance 4γ of R^γ.

Update procedure Since in the sliding window model points arrive one at a time and expire FIFO, the update procedure $\textsc{Update}_{6+\varepsilon}^{\mathrm{sw}}(p)$ (see Algorithm 14 for the pseudocode) is invoked once per arriving point p. For each guess $\gamma \in \Gamma$, it first removes from R_{orph}^γ all orphaned representatives that have expired, and, if the oldest attractor has expired, it calls DeleteAttraction to discard it. The latter procedure (Algorithm 15) turns the representative of the expiring attractor into an orphan, so that it remains available

to cover future points, and only then removes the attractor itself. The new point p is finally handed to INSERT (Algorithm 16).

INSERT(p,γ) computes the set D of attractors within distance 2γ of p. If $D = \emptyset$, then p is far from every attractor and becomes a new attractor, representing itself. If this makes $|A^\gamma| = k + 2$, the attractor with earliest deletion time is removed via DELETEATTRACTION. Moreover, whenever $|A^\gamma| = k + 1$, all orphaned representatives that expire before the earliest-expiring attractor are discarded: the $k + 1$ attractors already certify $r_k^*(X_{t'}) > \gamma$ for every t' until that attractor expires, so γ cannot be a useful guess in this interval and there is no need to keep points covering it. If instead $D \neq \emptyset$, then p, being the most recent point, becomes the new representative of every attractor in D.

Algorithm 14: $\textsc{Update}^{\text{sw}}_{6+\varepsilon}(p)$

```
1 foreach γ ∈ Γ do
2     foreach q ∈ R^γ_orph with t_del(q) ≤ t_arr(p) do
3         R^γ_orph ← R^γ_orph \ {q}
4     if ∃ a ∈ A^γ with t_del(a) ≤ t_arr(p) then
5         DeleteAttraction(a, γ)
6     Insert(p, γ)
```

Algorithm 15: DELETEATTRACTION(a, γ)

```
1 R^γ_orph ← R^γ_orph ∪ {rep^γ(a)};
2 R^γ_act ← R^γ_act \ {rep^γ(a)};
3 A^γ ← A^γ \ {a};
```

Algorithm 16: INSERT(p, γ)

```
D ← {a ∈ A^γ : d(p,a) ≤ 2γ};
if D = ∅ then
    A^γ ← A^γ ∪ {p};
    rep^γ(p) ← p;    R^γ_act ← R^γ_act ∪ {p};
    if |A^γ| = k + 2 then
        a_old ← arg min_{a∈A^γ} t_del(a);
        DELETEATTRACTION(a_old, γ);
    if |A^γ| = k + 1 then
        t_min ← min_{a∈A^γ} t_del(a);
        foreach q ∈ R^γ_orph with t_del(q) < t_min do
            R^γ_orph ← R^γ_orph \ {q}
else
    foreach a ∈ D do
        R^γ_act ← (R^γ_act \ {rep^γ(a)}) ∪ {p};
        rep^γ(a) ← p;
```

Query procedure To answer a query, the algorithm iterates through the guesses $\gamma \in \Gamma$ in increasing order and looks for the smallest γ such that $|A^\gamma| \le k$ and a 2γ-cover of R^γ of size at most k can be computed by the greedy strategy of Hochbaum and Shmoys [5]; the centers of this cover are returned as the solution. Since every active point is within 4γ of R^γ and every representative is within 2γ of a returned center, the triangle inequality yields a clustering of radius 6γ, while a guess for which more than k centers are required provides, by Proposition 2.1, a certificate that $r^*_k(X_t) > \gamma$.

Theorem 3.1 ([2])**.** *Given a stream with aspect ratio Δ and a window of size N, the algorithm described above maintains a $(6+\varepsilon)$-approximate solution for the metric k-center problem, storing at most $O((k/\varepsilon)\log\Delta)$ points, with an update time of $O(k^2\varepsilon^{-1}\log\Delta)$ per point.*

The space bound rests on the observation that each of A^γ, R^γ_{act} and R^γ_{orph} contains at most $k+1$ points, hence at most $3(k+1)$ points are kept for each of the $|\Gamma| = O(\varepsilon^{-1}\log\Delta)$ guesses. Crucially, this bound exploits the FIFO expiration order to argue that orphaned representatives cannot accumulate. In the next section we show how to lift this construction to the setting with lifetimes, where deletion times are arbitrary, recovering the same bound when deletions follow the arrival order.

3.2 With Lifetimes

The algorithm developed in the previous section achieves an almost-tight approximation ratio, but requires memory which is linear in the size of the active set. In this section, we devise an algorithm which requires much smaller memory, constant with respect to the size of the active set, at the cost of a slightly worse approximation guarantee.

The algorithm is based on the sliding-window algorithm of Cohen-Addad et al.[2] presented in Section 3.1. In the sliding window model, deletions occur in the same order as arrivals, which lets the algorithm discard expired attractors and orphaned representatives simply by scanning for the oldest ones. In the setting with lifetimes, instead, deletion times are arbitrary, and we generalize the construction so that the decision of which attractors and representatives to retain is driven by the (known) deletion times rather than by the arrival order. As we will see, the resulting data structure coincides with the one of Section 3.1 when points expire in FIFO order.

Let $\varepsilon > 0$ be a user-defined approximation tolerance, and let Γ be the set of guesses defined by Equation 2.1, instantiated now with $\beta = \varepsilon/6$. At any time t, the algorithm maintains, for each $\gamma \in \Gamma$, a data structure, generalizing the one of the sliding-window algorithm of Section 3.1, which is able to provide a solution of radius 6γ or a certificate that no solution of radius γ exists. The data structure consists of the sets:

- $A^\gamma = (a_1^\gamma, \ldots, a_{\ell^\gamma}^\gamma) \subseteq X_t$ a set of $\ell^\gamma \leq k+1$ *attractors* such that $d(a_i^\gamma, a_j^\gamma) > 2\gamma, \forall i \neq j$. For $x \in X_t$, $a \in A^\gamma$, we say that x is attracted by a if $t_{\text{arr}}(x) \geq t_{\text{arr}}(a)$ and $d(x,a) \leq 2\gamma$.

- $R_{\text{act}}^\gamma = (r_1^\gamma, \ldots, r_{\ell^\gamma}^\gamma) \subseteq X_t$ a set of *representatives*, where, for $1 \leq i \leq \ell^\gamma$, r_i is the point with latest deletion time among the ones attracted by $a_i \in A^\gamma$, and it is denoted with $rep^\gamma(a_i) = r_i$. We use $R_{\text{orph}}^\gamma \subseteq X_t$ to denote the set of representatives whose attractors are no longer in A^γ, and define $R^\gamma = R_{\text{act}}^\gamma \cup R_{\text{orph}}^\gamma$.

Observe that if $|A^\gamma| \geq k+1$, there are $k+1$ points at distance $> 2\gamma$, which implies that $\gamma < r_k^*(X_t)$. Conversely, if $|A^\gamma| \leq k$, we will show that any point of X_t is at distance at most 6γ from R^γ.

Update procedure When a new point p arrives, procedure $\textsc{Update}_{6+\varepsilon}(p)$ is called (see Algorithm 17 for the pseudocode) to handle the insertion of p, as well as possible deletions of expired points. For every $\gamma \in \Gamma$, the data structures are updated as follows. First, all points expired before time $t_{\text{arr}}(p)$ are removed from A^γ and R^γ. Then, if p

Algorithm 17: UPDATE$_{6+\varepsilon}(p)$

```
1  foreach γ ∈ Γ do
2      Remove points x of A^γ and R^γ with
         t_del(x) ≤ t_arr(p)
3      E ← {a ∈ A^γ : d(a,p) ≤ 2γ}
4      if E = ∅ then
5          A^γ ← A^γ ∪ {p}
6          rep^γ(p) ← p;  R^γ ← R^γ ∪ {p}
7          CLEANUP(p,γ)
8      else
9          E' ←
             {a ∈ E : t_del(rep^γ(a)) < t_del(p)}
10         if E' ≠ ∅ then
11             a ← arbitrary attractor in E'
12             R^γ ← R^γ \ {rep^γ(a)}
13             rep^γ(a) ← p;  R^γ ← R^γ ∪ {p}
```

Algorithm 18: CLEANUP(p, γ)

```
1  if |A^γ| = k + 2 then
2      a_min ←
         arg min_{a∈A^γ} t_del(a);
3      Remove a_min from A^γ;
4  if |A^γ| = k + 1 then
5      t_min ← min_{a∈A^γ} t_del(a);
6      Remove all points q
         with t_del(q) < t_min
         from R^γ
```

is at distance $> 2\gamma$ from every attractor $a \in A^\gamma$, then p is added to A^γ, and to R^γ as representative of itself (i.e., $p = rep^\gamma(p)$), and procedure CLEANUP(p,γ) is invoked to possibly evict unnecessary points from A^γ and R^γ. Otherwise, an attractor $a \in A^\gamma$ with $d(a,p) \leq 2\gamma$ and $t_{\text{del}}(rep^\gamma(a)) < t_{\text{del}}(p)$, if any, is arbitrarily chosen and p becomes the new representative $rep^\gamma(a)$. If no such attractor is found, p is discarded.

Procedure CLEANUP(p,γ) (see Algorithm 18 for the pseudocode), works as follows. If $|A^\gamma| = k + 2$, then the attractor with minimum deletion time is removed from the set. After this removal, or if the size of A^γ is $k + 1$ at the start of the procedure, all representatives q such that $t_{\text{del}}(q) < t_{\min}$, where $t_{\min} = \min_{a \in A^\gamma} t_{\text{del}}(a)$ is the minimum deletion time of an attractor, are removed from R^γ. These removals are justified by the following argument. If $|A^\gamma| = k+1$, then these $k+1$ points, which are at distance greater than 2γ from one another, provide a certificate that no solution of radius $\leq \gamma$ exists for any $X_{t'}$ with $t \leq t' < t_{\min}$, hence γ will not be a useful guess until $t_{\min}$, and there is no need to keep points that expire in this time interval.

Query procedure

At any time t, to obtain a solution to k-center for the active set X_t, procedure QUERY$_{6+\varepsilon}(t)$ is invoked (see Algorithm 19). First, for each $\gamma \in \Gamma$, all points p with $t_{\text{del}}(p) \leq t$ are removed from A^γ and R^γ. Then, the procedure searches the minimum guess $\gamma \in \Gamma$ such that $|A^\gamma| \leq k$ and a 2γ-covering C^γ of R^γ of size at most k can be found using the simple greedy strategy by [5], and this set C^γ is returned as the solution.

Algorithm 19: $\textsc{Query}_{6+\varepsilon}(t)$

```
foreach γ ∈ Γ do
    Remove points x of A^γ and R^γ with t_del(x) ≤ t
for increasing values of γ ∈ Γ such that |A^γ| ≤ k do
    C ← ∅;
    foreach q ∈ R^γ do
        if C = ∅ or d(q, C) > 2γ then C ← C ∪ {q} ;
        if |C| > k then go to next γ;
    return C
```

3.2.1 Approximation

In what follows, we denote with superscript t the states of data structures either after the call of $\textsc{Update}_{6+\varepsilon}(p)$, with $t_{\text{arr}}(p) = t$, or after the execution of the first for loop in $\textsc{Query}_{6+\varepsilon}(t)$. To prove the approximation ratio, we first prove that the algorithm maintains the following invariant.

Lemma 3.1. *After any execution of* $\textsc{Update}_{6+\varepsilon}(p)$*, the following properties hold for each* $\gamma \in \Gamma$*, and* $t = t_{\text{arr}}(p)$ *:*

1. *If* $|A^{\gamma,t}| \leq k$*, then* $\forall x \in X_t$ *we have* $d(x, R^{\gamma,t}) \leq 4\gamma$*.*
2. *If* $|A^{\gamma,t}| = k+1$*, then* $\forall x \in X_t$ *such that* $t_{\text{del}}(x) \geq \min_{a \in A^{\gamma,t}} t_{\text{del}}(a)$*, we have* $d(x, R^{\gamma,t}) \leq 4\gamma$*.*

Proof. Let us focus on an arbitrary guess γ. We say that a point $q \in X_t$ is *relevant for* γ *and* t if $|A^{\gamma,t}| \leq k$ or $|A^{\gamma,t}| > k$ and $t_{\text{del}}(q) \geq \min_{a \in A^{\gamma,t}} t_{\text{del}}(a)$. Clearly, the proof concerns only relevant points. Also, for every point $q \in X_t$, we let $\psi^\gamma(q)$ denote the attractor for which q was selected as representative by $\textsc{Update}_{6+\varepsilon}(q)$ (possibly, q itself) or any attractor belonging to the set E during the execution of $\textsc{Update}_{6+\varepsilon}(q)$. Clearly, $d(q,\psi^\gamma(q)) \leq 2\gamma$. We will argue that for every point $q \in X_t$ which is relevant for γ and t, there exists a point $r \in R^{\gamma,t}$ such that $\psi^\gamma(r) = \psi^\gamma(q)$.

If q is the point arrived at time t (i.e., $q = p$ and $t_{\text{arr}}(q) = t$) then we set $r = rep^{\gamma,t}(\psi^\gamma(q))$, which is possibly q itself. If q arrived at an earlier time, it is immediate to argue that since q is relevant at time t, it has also been relevant at all times t', with $t_{\text{arr}}(q) \leq t' < t$, when a point arrived or a query was invoked. Set $r_0 = rep^{\gamma,t_{\text{arr}}(q)}(\psi^\gamma(q))$. We have that $\psi^\gamma(r_0) = \psi^\gamma(q)$. Also, it holds that $t_{\text{del}}(r_0) \geq t_{\text{del}}(q)$, therefore r_0 is also

relevant at all times t', with $t_{\text{arr}}(q) \leq t' < t$. Being relevant, r_0 has never been removed from $R^{\gamma,t'}$ during some invocation of CLEANUP, which only removes non-relevant points. Consequently, since r_0 is in $R^{\gamma,t_{\text{arr}}(q)}$, either r_0 is still in $R^{\gamma,t}$ (and we set $r = r_0$), or it has been expunged from some set $R^{\gamma,t_{\text{arr}}(r_1)}$, due to the arrival of some longer-lived representative r_1 at time $t_{\text{arr}}(r_1) > t_{\text{arr}}(q)$. Observe that $\psi^\gamma(r_1) = \psi^\gamma(r_0)$, and that r_1 is also relevant at time t. If $r_1 \in R^{\gamma,t}$, we set $r = r_1$, otherwise, we apply the same reasoning to r_1. Iterating the argument, we determine a sequence of relevant points $r_0, r_1, \ldots, r_h$, with $t_{\text{arr}}(q) \leq t_{\text{arr}}(r_0) < t_{\text{arr}}(r_1) \ldots < t_{\text{arr}}(r_h)$ and $\psi^\gamma(q) = \psi^\gamma(r_0) = \ldots = \psi^\gamma(r_h)$, with $r_h \in R^{\gamma,t}$. We then set $r = r_h$. The lemma follows immediately since $\psi^\gamma(r) = \psi^\gamma(q)$ implies $d(q,r) \leq 4\gamma$. □

Based on the invariants, we can prove the approximation ratio of the solutions returned by $\text{QUERY}_{6+\varepsilon}$.

Theorem 3.2. *For any time t, the solution returned by $\text{QUERY}_{6+\varepsilon}(t)$ is a $(6+\varepsilon)$-approximation for the k-center problem on the active set X_t.*

Proof. Let $\gamma = (1+\beta)^i \in \Gamma$ be the guess such that the returned solution C is computed from $R^{\gamma,t}$, for some $\lfloor \log_{1+\beta} d_{\min} \rfloor \leq i \leq \lceil \log_{1+\beta} d_{\max} \rceil$. By construction, $|A^{\gamma,t}| \leq k$, and, for each $p \in R^{\gamma,t}$, $d(p,C) \leq 2\gamma$. Using Lemma 3.1 and the triangle inequality, we have that for each $p \in X_t$, $d(p,C) \leq 6\gamma$. Now, if $i = \lfloor \log_{1+\beta} d_{\min} \rfloor$ then $\gamma \leq r_k^*(X_t)$ and the theorem holds. Instead, if $i > \lfloor \log_{1+\beta} d_{\min} \rfloor$, we have, by construction, that for $\gamma' = (1+\beta)^{i-1} = \gamma/(1+\beta)$ either $|A^{\gamma',t}| = k+1$ or $k+1$ points of $R^{\gamma',t}$ have been found at distance $> 2\gamma'$ from one another. From what we observed earlier, in either case we have $\gamma' < r_k^*(X_t)$, hence $\gamma < (1+\beta) r_k^*(X_t)$. Thus, for each $p \in X_t$, $d(p,C) \leq 6\gamma < 6(1+\beta) r_k^*(X_t) \leq (6+\varepsilon) r_k^*(X_t)$, and the theorem follows. □

3.2.2 Running time and memory complexity

We now analyze the number of points maintained by the data structure employed by our algorithm, which directly influences the time and memory requirements. The analysis is made parametric with respect to a specific notion of ordering of the input dataset X, which is defined below and it is a crucial novel ingredient introduced by this thesis.

Definition 3.1. *For an integer parameter $H \geq 0$, we say that the set X is* H-ordered *if for every pair of points $p,q \in X$ such that $t_{\text{arr}}(p) < t_{\text{arr}}(q)$ and $|\{x \in X\colon\ t_{\text{arr}}(p) < t_{\text{arr}}(x) < t_{\text{arr}}(q)\}| \geq H$, i.e., at least H points arrive between p and q, we have that $t_{\text{del}}(p) < t_{\text{del}}(q)$.*

Intuitively, the above notion provides a quantitative characterization of how arbitrary the points' arrival and departure times are, and a low value of H means that points are deleted roughly in the order in which they arrive. For example, in the sliding-window setting, or in general when points are deleted in the order they arrive, the pointsets are 0-ordered. The assumption of H-ordered sets is realistic: consider for example a stream with batch processing (e.g., buffers in TCP packet routers, or intermediate storage for industrial manufacturing). Items arrive in sequence and are grouped into batches of size H, with each batch being processed as a unit. While the deletion order of items within a batch may change, batches themselves are served in arrival order, resulting in an H-ordered set.

Importantly, the algorithm does not require the knowledge of H, which is only used by the analysis. Indeed, under H-ordered sets, we have the following result.

Lemma 3.2. *Suppose that X is H-ordered, for some $H \geq 0$. Then, for any guess $\gamma \in \Gamma$ and time t,*

$$|A^{\gamma,t}| + |R^{\gamma,t}| = O(\min\{(k+H), |X_t|\}) .$$

Proof. Let us fix arbitrary γ and t, and observe that, straightforwardly, $|A^{\gamma,t}| + |R^{\gamma,t}| \leq 2|X_t|$. We will now show that $|A^{\gamma,t}| + |R^{\gamma,t}| = O(k+H)$. By construction, we know $|A^{\gamma,t}| \leq k+1$, so we only have to show that $|R^{\gamma,t}| = O(k+H)$. Let $A^{\gamma}_{\text{all}}(t)$ denote the set of all attractors ever arrived up to time t, and let q be the one with largest arrival time among those not in $A^{\gamma,t}$, namely, $q = \arg\max_{a \in A^{\gamma}_{\text{all}}(t) \setminus A^{\gamma,t}} t_{\text{arr}}(a)$. Hence, q arrived at some time $t_{\text{arr}}(q) < t$ and was removed from A^{γ} at some time t', with $t_{\text{arr}}(q) \leq t' < t$, either by Line 2 of the UPDATE procedure, because it expired, or by Line 3 of the CLEANUP procedure. In the time interval $(t_{\text{arr}}(q), t]$ no more than $k+1$ new attractors arrived since, otherwise, one of the attractors arrived in $(t_{\text{arr}}(q), t]$ would not belong to $A^{\gamma,t}$, because A^{γ} can never contain more than $k+1$ points, and this would contradict the choice of q. For a point $p \in R^{\gamma,t}$, let $\psi^{\gamma}(p)$ denote the attractor for which p is representative. We partition $R^{\gamma,t}$ into 3 subsets:

- $R^{\gamma,t}(1) = \{p \in R^{\gamma,t} : \ t_{\text{arr}}(\psi^{\gamma}(p)) \in (t_{\text{arr}}(q), t] \ \}$
- $R^{\gamma,t}(2) = \{p \in R^{\gamma,t} : \ \psi^{\gamma}(p) \in A^{\gamma, t_{\text{arr}}(q)} \ \}$
- $R^{\gamma,t}(3) = \{p \in R^{\gamma,t} : \ \psi^{\gamma}(p) \notin A^{\gamma, t_{\text{arr}}(q)} \text{ and } t_{\text{arr}}(\psi^{\gamma}(p)) < t_{\text{arr}}(q) \ \}$

It is immediate to see that $R^{\gamma,t} = R^{\gamma,t}(1) \cup R^{\gamma,t}(2) \cup R^{\gamma,t}(3)$. Moreover, $|R^{\gamma,t}(1)| \leq k+1$ from what we observed before, and $|R^{\gamma,t}(2)| \leq k+1$ since $|A^{\gamma, t_{\text{arr}}(q)}| \leq k+1$ by construction. We now argue that $|R^{\gamma,t}(3)| = O(H)$. First note that all points of $R^{\gamma,t}(3)$

must have arrived prior to $t_{\text{arr}}(q)$, since, by definition of $R^{\gamma,t}(3)$, the attractors for which they are representatives, entered and left A^γ before $t_{\text{arr}}(q)$. Let p be the point of $R^{\gamma,t}(3)$ with earliest arrival time. So, all other points of $R^{\gamma,t}(3)$ must have arrived in the open interval $(t_{\text{arr}}(p),t_{\text{arr}}(q))$. Thus, to show that $|R^{\gamma,t}(3)| = O(H)$ it is sufficient to argue that less than H points arrived in $(t_{\text{arr}}(p),t_{\text{arr}}(q))$. By contradiction, suppose that this is not the case. Then, the assumption that X is H-ordered implies that $t_{\text{del}}(p) < t_{\text{del}}(q)$. Since q was removed from A^γ at time $t' < t$, we have two cases:

- q expired at t' and was removed at Line 2 of the UPDATE procedure. Hence, $t_{\text{del}}(p) < t_{\text{del}}(q) = t' < t$, and p cannot be in $R^{\gamma,t}$.
- q was removed from A^γ at time t' at Line 3 of the CLEANUP procedure, since q had minimum deletion time among $k+2$ attractors. Then, right after the removal of q from A^γ, $t_{\min}$ was set to a value $\geq t_{\text{del}}(q)$ and all representatives r with $t_{\text{del}}(r) < t_{\text{del}}(q) \leq t_{\min}$ were removed. Therefore, since $t_{\text{del}}(p) < t_{\text{del}}(q)$, then p would have been removed as well.

In either case, p could not be in $R^{\gamma,t}$, which contradicts the choice of p. □

The next theorems, based on Lemma 3.2, state upper bounds for the update and query running times, and for the memory complexity. Let $\text{pr}(t)$ be the time of the last update/query operation before t.

Theorem 3.3. $\text{UPDATE}_{6+\varepsilon}(p)$ *requires* $O\big((\log\Delta/\varepsilon)\min\{(k+H),|X_{\text{pr}(t_{\text{arr}}(p))}|\}\big)$ *operations.*

Theorem 3.4. $\text{QUERY}_{6+\varepsilon}(t)$ *requires* $O\big((\log\Delta/\varepsilon)k\cdot\min\{(k+H),|X_{\text{pr}(t)}|\}\big)$ *operations.*

Theorem 3.5. *The maximum number of points maintained in memory when processing the pointset X is* $O((\log\Delta/\varepsilon)\min\{(k+H),\max_t|X_t|\})$.

Note that, for $H \in O(k)$, update times and the memory complexity are bounded by $O((\log\Delta/\varepsilon)k)$.

3.2.3 Comparison with related work

The memory and time complexities of the algorithm presented in this chapter depend on the tameness of the update sequences. Importantly, the memory usage never exceeds the one of the $(2+\varepsilon)$-approximation algorithm presented in Section 2.3, and, for small H (e.g., $H \in O(k)$), it is significantly smaller memory, while attaining comparable update time.

Moreover, unlike any existing fully-dynamic algorithm in general metrics, the bound on the operations is *worst-case* rather than amortized. These gains, however, come at the cost of a (slightly) worse approximation ratio.

Our setting can be viewed as a generalization of the sliding window one. Indeed, our algorithm extends the sliding-window one of [2], described in Section 3.1, by allowing arbitrary deletions. Despite the increased generality, we match the same memory and time complexities when $H = 0$. As discussed in Section 2.4.1, [6] present an algorithm for k-center in a setting akin the one with lifetimes, which stores $O((\log \Delta/\varepsilon)\, k^3)$ points regardless of H-ordering and is more memory-efficient than ours for large H (i.e., $H \in \Omega(k^3)$). However, their algorithm has expensive update times, and, more importantly, a worse non-constant approximation ratio, growing linearly with k.

Chapter 4

Experiments

In this chapter we show the empirical experiments we did on the previously presented algorithms. All the experiments were ran on a laptop with 8 core AMD Ryzen 7 5800U processor and 16GB of RAM. You can find all the source code used on github [7].Of all the shown algorithms we made our own implementation. Only exceptions are Chan's algorithm, of which we also picked the original implementation of the authors [8], and the Cohen-Addad's one, which we did not include as it is just a special case of our low memory algorithm and no public implementation was made available by the authors.

We did make use of generative AI to help with the process of writing code, although we tried our best to not blindly trust the models on everything. Specifically:

- All algorithms were first implemented from scratch by hand and then fed to Claude for debugging.
- The python scripts for coordinating the experiments, generating the datasets and plotting the results were generated by Claude and reviewed afterwards.

In the experiments we mix fully dynamic algorithms with algorithms with lifetimes and, for simplicity, we feed all algorithms the same datasets. This means that fully dynamic algorithms will read in input the deletion times, but they discard it immediately. Furthermore, algorithms with lifetimes will have access to deletions in the stream the moment they happen, but they will ignore them as in the model with lifetimes only insertions should be streamed. We divided the measured execution time in **update** and **query time**, we also left the input streaming out of the measured time, in order to more accurately measuring the difference between implementations. We also kept track of the **maximal number of saved points**, for evaluating our $(6 + \varepsilon)$-approximation.

We will refer to the algorithms by the following names:

- **gonzalez**: Gonzalez' 2-approximation, described in Chapter 1. This algorithm is included as a sanity check for empirically estimating the approximation of our implementations and it is not modified for cleverly handling insertions or deletions.

- **chan-original**: Chan's fully dynamic algorithm described in section 2.1.

- **bateni**: Our implementation of Bateni's fully dynamic algorithm described in section 2.2 (no implementation was provided by the authors).

- **ours2eps**: Our $(2+\varepsilon)$-approximation algorithm with lifetimes, described in section 2.3.

- **ours2epsrandom**: The fully dynamic random variant of our $(2+\varepsilon)$-approximation algorithm, as described in 2.4.

- **ours6eps**: Our $(6+\varepsilon)$-approximation algorithm with low memory described in section 3.2.

We compare our algorithms on 2 types of datasets:

- **random**: In this dataset points are picked from a uniform distribution in $[0,1]^d$. Insertion and deletion times are picked randomly with constraints as to satisfy the maximal number of active points and possible H-orderdness of the dataset.

- **gowalla**: This is a real life dataset taken from SNAP [9]. It represents a social network in which people can share with others information on their real life locations by check-ins. We are not interested in the network of user friendships, rather than on the check-ins of each user. Each check-in has a clear timestamp, along with latitudinal and longitudinal coordinates of the location the user was when it took the check-in. The deletion time of each check-in is represented by the following time the user made another check-in (the last check-in is discarded). You can look at the following table to see an example of the raw data:

User	Check-in Time	Latitude	Longitude
196514	2010-07-24T13:41:10Z	53.364905	-2.270824
196514	2010-07-24T13:43:18Z	53.3679640626	-2.2792943689
196514	2010-07-24T13:44:08Z	53.3675663377	-2.278631763
196514	2010-07-24T13:44:26Z	53.3674087524	-2.2783813477
196514	2010-07-24T13:44:38Z	53.3663709833	-2.2700764333
196514	2010-07-24T13:44:46Z	53.3653895945	-2.2754087046
196514	2010-07-24T13:44:58Z	53.360511233	-2.276369017
196514	2010-07-24T13:45:06Z	53.3648119	-2.2723465833

Table 4.1: Example of user check-ins in Gowalla dataset.

Given that coordinates are on a spherical surface we use the great-circle distance as our metric for this dataset, whilst the other two are Euclidean.

We will investigate the impact on the performance of many different parameters, the following table is a recap of what each parameter is, along with its default value if not otherwise specified:

Parameter	Description	Default Value
N	Total number of insertions	10000
`max_n_active`	Maximal number of active points	1000
d	Dimensionality of the points	3
k	Number of centers	10
ε	Approximation accuracy parameter	0.5
H	H-ordering of the dataset, as defined in Section 3.2	∞

Table 4.2: Parameter Descriptions

4.1 Approximation

We will now show the comparison of the algorithms against Gonzalez' 2-approximation in order to check wether we made a major implementation mistake. We are forced to use Gonzalez' approximation instead of finding the optimal radius as it will be unfeasible to compute for a sensible number of active points and k (a solution which checks every possible center selection runs in $\Omega\left(|X_t|^k\right)$). Even with the Gonzalez' algorithm we are limited to a low number of active points, because each query requires $\Omega\left(n \cdot k\right)$ to run.

The comparison is made by running all algorithms in the exact same datasets with the same queries. Afterwards, for each query, the radius induced by the centers found by Gonzalez is taken as a baseline for the optimal solution and is compared with the other algorithms. We test the algorithms across various values of k on the random and Gowalla datasets. We fix $\varepsilon = 0.5$. For each run (i.e. each different pair of k and dataset), we randomly ask a query after each insertion with 5% probability. The plots show the average ratio of the radius found by each algorithm and the radius found by Gonzalez across all queries made in one run.

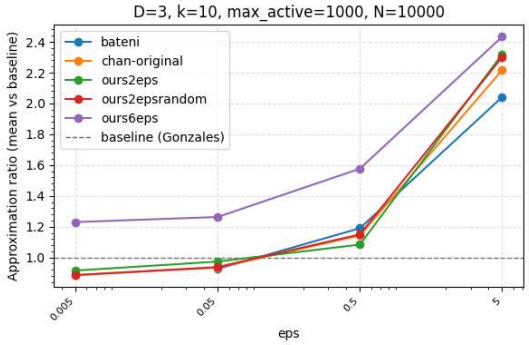

Figure 4.1: random

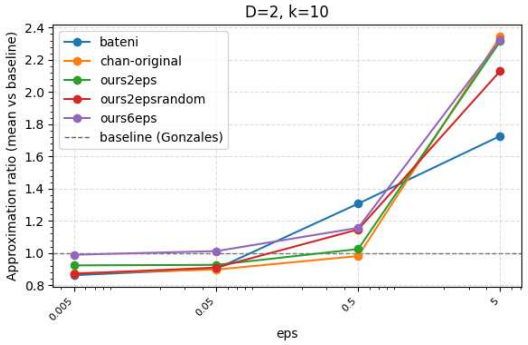

Figure 4.2: gowalla

We can see that all $(2+\varepsilon)$-approximation algorithms exhibit a clear degradation of the approximation factor as ε increases as expected. The found solution is even better than Gonzalez' for $\varepsilon = 0.05$. For higher values of ε, if we assume that Gonzalez found a 2-approximation of the optimal solution and therefore we multiply the average ratio by 2, we do not encounter an approximation higher than $2+\varepsilon$ for all $(2+\varepsilon)$-approximation algorithms, with just one exception made by Bateni in the Gowalla dataset with $\varepsilon = 0.5$. The exception is almost surely due to the fact that Gonzalez can still find a solution more precise than a worse possible 2-approximation and it isn't strong enough to conclude that Bateni's algorithms might produce an approximation worse than $(2+\varepsilon)$.

In the random dataset, the $(6+\varepsilon)$-approximation, while being clearly worse than the counterparts, doesn't come near the $6+\varepsilon$ mark. For example, with $\varepsilon = 0.2$, we obtain a 1.4 ratio with respect to Gonzalez, which is less than a $2.8 < 6.2$ approximation of the optimal radius. Moreover, in the Gowalla dataset the $(6+\varepsilon)$-approximation is nearly indistinguishable from the other algorithms. This means that $6+\varepsilon$ is a purely theoretical upper bound which can be much lower in practice.

4.2 Running Time

Here we evaluate the timing performance of the algorithms starting with the update time.

We start by looking at the impact of parameters ε and d. Both plots are scaled logarithmically on both axes. We consider the random dataset.

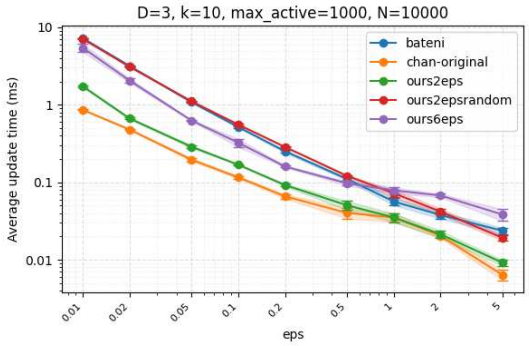

Figure 4.3: Plot over ε

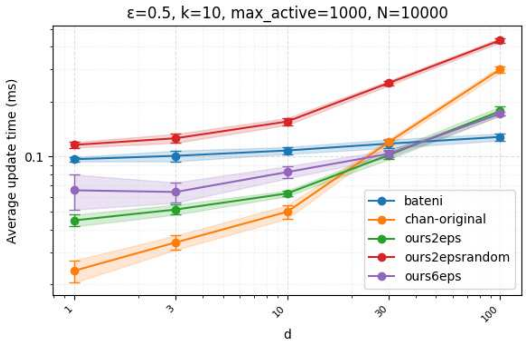

Figure 4.4: Plot over d

The plot of ε shows exactly what we expected: the running time decreases with ε. The one of d, whilst still having an upwards trend across all the algorithms as expected, allows us to guess the proportion of the time each algorithm dedicates to computing distances. This is due to the fact that the only part in which d influences the running time is inside the distance calculation, which is the same function across all algorithms. Bateni shows an almost flat graph, indicating that distance computations do not occupy the majority of the time, probably due to the priority queue updates at each guess. The algorithm from Chan is the most efficient in terms of non-distance computations, as we can see by its low running time when $d = 3$, but pays the $O(k^2)$ distance computed per-update when d gets large enough.

We now show the difference in runtime across multiple values of k. As before, plots are in loglog scale. We will look at both the random and Gowalla dataset to also get a sense of how well the algorithm run on real world data.

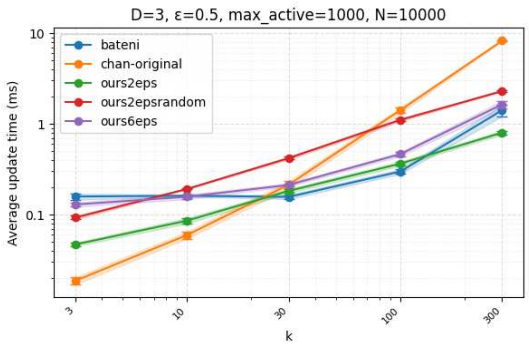

Figure 4.5: Random dataset

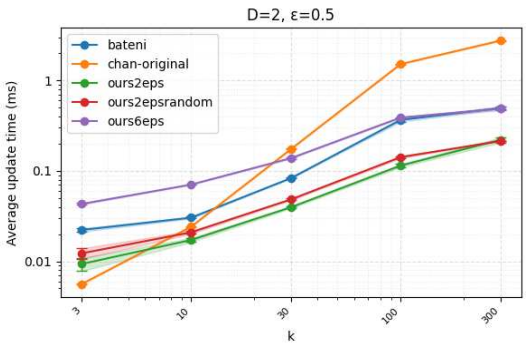

Figure 4.6: Gowalla dataset

In both figures we can visually see the quadratic dependency on k of Chan's algorithm, which in loglog plot is indicated by a steeper line. Bateni's algorithm has a weird behaviour on the fully random dataset, as it seems to first lower the update time up to $k = 30$ before finally starting to grow. This behaviour is probably due to the geometrical disposition of the random point cloud and might need a complicated and ultimately useless analysis to justify. What's interesting is that both our lifetimes algorithm and its fully dynamic counterpart perform really well on real world data, despite showing no practical improvement on random data against the state of the art.

The last parameter we want to control is the number of active points, given that the theoretical improvement of our algorithm against Bateni's stands from the removal of a multiplicative logarithm in this number. For this experiment we will only compare our $(2+\varepsilon)$-algorithms against Bateni and we will do so in the random dataset, which lets us control the maximal number of active points as oppose to the Gowalla dataset. Given that we might need to have an high number of active points for this experiment we set $n = 300000$. The plots are only logarithmic in the x-axis.

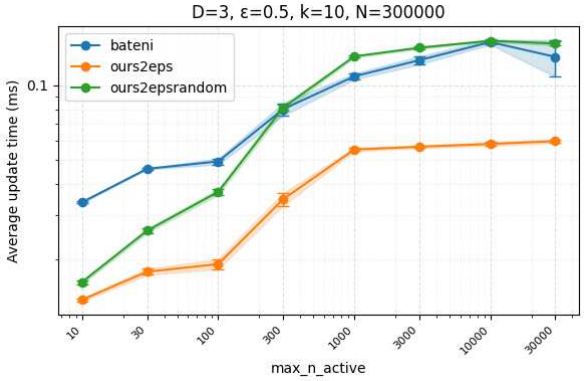

Figure 4.7: Plot over `max_n_active`

While we see a better performance of our algorithm with lifetimes against Bateni, the picture is not clear enough. We see from all algorithm an increase of runtime from 100 to 1000 active points, for it to remain almost constant in the remaining part of the plot. The same behaviour was observed with other values of N and k, and it is probably due to the random dataset not being able to produce an adversarial sequence of operations that make Bateni run in the worst possible running time. We could try to find an ad-hoc adversarial dataset, but that would be beyond the scope of the thesis.

To conclude this section, we analyze the query time. We mainly do this analysis as our $(6+\varepsilon)$-approximation is influenced by parameters k and H in the query cost. Therefore, we look at how query time behaves across both k and H on the random dataset.

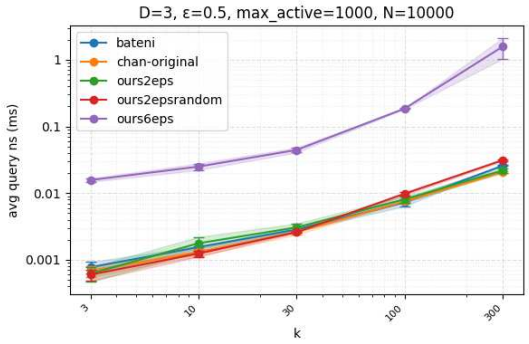

Figure 4.8: Query time over k

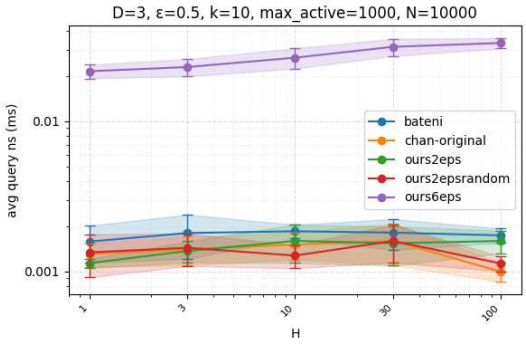

Figure 4.9: Query time over H

The $(6+\varepsilon)$-approximation exhibits a clear difference in query time with respect to the others, this is due to the fact that the query algorithm needs to find a clustering across all representative and does not maintain k centers per guess like the other algorithms. We can see an increase of query cost with respect to k across all algorithms, as the k centers still have to be printed in output. The H value does not affect the $(2+\varepsilon)$-approximations at all as expected, the only affected algorithm is the $(6+\varepsilon)$ one, with a slight increase in query time.

4.3 Memory Consumption

In the analysis of our $(6+\varepsilon)$-approximation the notion of H-ordering comes up. This brief section aims to find if the dependency on the H-ordering of the dataset on the number of saved points is purely theoretical or has some real impact in the number of saved points. We ran our experiment on the random dataset across different values of H and k, here we report the maximal total number of points saved by the algorithm.

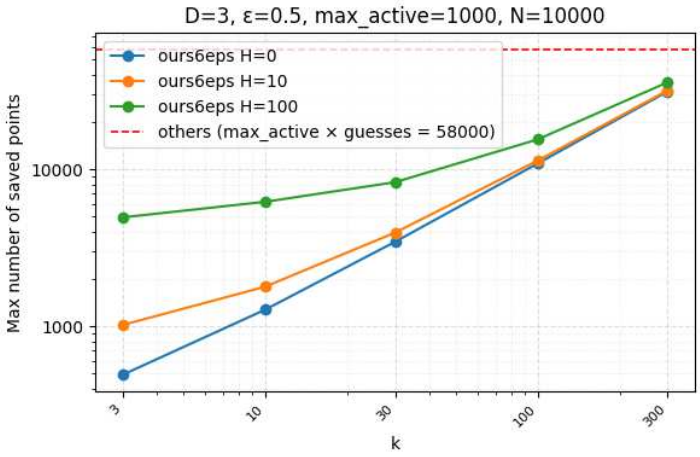

Figure 4.10: Memory consumption over k

We see that H has a tangible additive impact on the number of saved points, confirming theoretical analysis. We also see a linear increase in points saved with respect to k, but this was obvious by the fact that we need to save up to $k+1$ attractors to certify the unfeasibility of a guess. It remains unclear wether this dependency on H is due only to the algorithm in question or if it exists an information theoretical lower bound that proves this for any algorithm, further research in this direction is needed.

4.4 Conclusions

The theoretical advantage of utilizing our algorithms when the knowledge of lifetimes is available is undisputable: we obtain a $O(\log(n))$ reduction on the running time of the $(2+\varepsilon)$-approximation and set a new baseline for the low memory case, as no algorithms achieved a $6+\varepsilon$ approximation outside the sliding window model.

For the practicality of our approach the picture is not so decisive, but it is still in our favour, at least on some cases.

As of these implementations, the algorithm from Chan et al. seems to perform better than the counterparts for small values of k. This can be inherent to the additional complexity of needing to handle persistent and vanishing points, but we suspect that the original implementation of the authors might simply be better than what we came up with. Moreover, for not so high values of k (e.g. 30) the algorithm becomes worse than the others.

Across all the experiments, the algorithm from Bateni et al. managed to outperform us only on the random dataset with a really high value of d and matched our performances in the random dataset with values of k between 30 to 100. Moreover, in the real world case the performance of Bateni stays consistently above both our algorithm with lifetimes and our fully dynamic variant.

The $(6+\varepsilon)$-approximation also seems to perform well on real world data, maintaining an approximation on par with every other algorithm, making it competitive even against theoretically better algorithms.

This means that the algorithms we developed have not only good theoretical guarantees, but promise better performances on practical applications and could be used in real world scenarios to do fast data analysis.

Bibliography

[1] T. F. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theoretical Computer Science*, vol. 38, pp. 293–306, 1985.

[2] V. Cohen-Addad, C. Schwiegelshohn, and C. Sohler, "Diameter and k-center in sliding windows," in *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2016, pp. 19–1.

[3] T. H. Chan, A. Guerqin, and M. Sozio, "Fully dynamic k-center clustering," in *World Wide Web Conference (WWW)*, 2018, pp. 579–587.

[4] M. Bateni, H. Esfandiari, H. Fichtenberger, *et al.*, "Optimal fully dynamic k-center clustering for adaptive and oblivious adversaries," in *Symposium on Discrete Algorithms (SODA)*, 2023.

[5] D. S. Hochbaum and D. B. Shmoys, "A unified approach to approximation algorithms for bottleneck problems," *Journal of the ACM (JACM)*, vol. 33, no. 3, pp. 533–550, 1986.

[6] L. Blank, S. Cabello, M. Hajiaghayi, *et al.*, "The general expiration streaming model: Diameter, k-center, counting, sampling, and friends," *arXiv preprint arXiv:2509.07587*, 2025.

[7] S. Moretti, *Experiments included in this thesis*, `https://github.com/simpatine/dynamic-k-center-lifetimes`, Accessed: 2026-07-01.

[8] T. H. Chan, A. Guerqin, and M. Sozio, *Fully dynamic k-center clustering experiments' github page*, `https://github.com/fe6Bc5R4JvLkFkSeExHM/k-center`, Accessed: 2026-07-01.

[9] E. Cho, E. Myers, and J. Leskovec, *Snap: Gowalla dataset*, `https://snap.stanford.edu/data/loc-Gowalla.html`, Accessed: 2026-06-27.