Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

University of Padua

INFORMATION ENGINEERING DEPARTMENT (DEI)

MASTER'S DEGREE IN
COMPUTER ENGINEERING

Cover tree based dynamization of clustering algorithms

Supervisors:

Prof. Andrea Pietracaprina
Prof. Geppino Pucci

Candidate:

Trevisan Thomas
2054140

Academic Year: 2022/2023

Abstract

In this work, cover trees are the main focus, and they serve as a data structure to efficiently store metric data. We utilize them for dynamically handling the k-center problem, both with and without outliers. The cover tree data structure is designed to retrieve a coreset, a very succinct summary of the data, which is then fed to an offline clustering algorithm to quickly obtain a solution for the whole dataset.

With respect to the original definition, the cover tree implemented is augmented, to maintain additional information crucial for extracting reasonable coresets. The solutions obtainable for the mentioned problems are $(\alpha+\varepsilon)$-approximations, where α represents the best-known approximation achievable in polynomial time in the standard offline setting, and $\varepsilon > 0$ is a user-provided accuracy parameter.

The main objective in using a dynamic data structure is to obtain a reasonable solution, in comparison to the solution obtained by applying the clustering algorithms from scratch to all the data points. To ascertain the quality of our solution, we conduct a series of experiments to evaluate its performance and to fine-tune the involved parameters.

Abstract

In questo lavoro, i Cover Tree sono l'obiettivo principale e servono come struttura di dati per memorizzare in modo efficiente i dati. Li utilizziamo per gestire dinamicamente il k-Center problem, sia con che senza outlier. La struttura Cover Tree è progettata per recuperare un coreset, una rappresentazione molto piccola dei dati, che viene poi fornito a un algoritmo di clustering offline per ottenere rapidamente una soluzione per l'intero set di dati.

Rispetto alla definizione originale, il Cover Tree implementato viene aumentato con nuovi campi, per mantenere informazioni aggiuntive cruciali per l'estrazione di coreset ragionevoli. Le soluzioni ottenibili per i problemi citati sono approssimazioni $(\alpha + \varepsilon)$, dove α rappresenta la migliore approssimazione nota ottenibile in tempo polinomiale nell'impostazione standard offline e $\varepsilon > 0$ è un parametro di precisione fornito dall'utente.

L'obiettivo principale dell'utilizzo di una struttura dati dinamica è quello di ottenere una soluzione ragionevole, rispetto a quella ottenuta applicando gli algoritmi di clustering da zero a tutti i dati. Per verificare la qualità della nostra soluzione, conduciamo una serie di esperimenti per valutarne le prestazioni e mettere a punto i parametri coinvolti.

Contents

Chapter 1

Introduction

Clustering is a widely used data analysis primitive in pattern recognition, statistics and machine learning. This problem aims to partition a large set of objects or points, typically represented as vectors, based on predefined similarity metrics. Points within the same cluster are expected to be more similar to each other than to any points from other clusters. Conversely, points from different clusters are expected to be more dissimilar [1].

Data clustering techniques can be classified in two categories: hierarchical and partitional. Hierarchical algorithms find successive clustering using previously found clusterings (merging two clusters to create a new one, or dividing one cluster into two), while partitional algorithms determine all clusters simultaneously. Although hierarchical clustering allows us to select the number of clusters that fits the input dataset best, as it produces a set of clustering solutions in a hierarchical representation, partitional clustering has lower time complexity and it is simpler. In this thesis, we approach the clustering problem using partitional algorithms.

The problem that we want to solve in our work is the k-Center problem, which is a center-based clustering. In this class of clustering problems, each cluster has a central representative point, called center. The goal of the k-Center problem is to find k centers that minimize the maximum distance to their closest data point. The points in the input set will be then assigned to the cluster having the closest center. An important variant of this problem, introduced by Charikar et al. [2], is the k-Center problem with z outliers, wherein the clustering process is permitted to disregard z data points from the input.

The clustering problem can be further divided into two categories depending on the input dataset: clustering can be either static or dynamic. Static clustering assumes a fixed dataset that does not change during the clustering process. This approach works well with stable datasets that undergo minimal evolution. It offers lower time complexity compared to the dynamic approach and allows optimization of the clustering solution for all data points in one run. Static clustering finds applications in customer segmentation, image segmentation, and document clustering. In such cases, significant data changes are not expected, so this approach requires few iterations and performs well with relatively small, well-structured datasets.

In contrast, dynamic clustering deals with data that evolves or changes over time. This may involve datasets where one or more points are inserted, deleted or updated (equivalent to deleting and inserting a new point). The strength of the dynamic approach is that it doesn't rebuild the clustering from scratch every time the data changes. Instead, dynamic clustering updates the existing clusters to accommodate new data or changes in existing data points. However, this leads to high computational complexity. Dynamic clustering finds applications in network traffic analysis, sensor data processing, online retail recommendations, and anomaly detection [3].

In the field of machine learning, clustering is often performed on datasets with increasing dimensionality and scale. Trying to process all the points can result in significant computational efforts with poor results. If we are in a dynamic environment, performing clustering multiple times using a static approach whenever the dataset changes becomes infeasible due to the computational load. On the other hand, for large datasets, even a fully dynamic algorithm becomes infeasible [4].

The approach used in this work involves extracting representative subsets from the entire set of points, commonly referred to as "coresets", which retain essential information while significantly reducing the input size.

Specifically, we insert all the dataset's points into a data structure called Cover Tree, as defined by Beygelzimer et al. in [5]. By leveraging the hierarchical structure of the tree and the properties it possesses, we extract a coreset from it. The properties of the Cover Tree, as well as the manner in which the points are stored, are based on the same similarity metrics used in the clustering problem we aim to solve. Consequently, the coreset retrieved serves as a robust representation of the dataset.

Dealing with a small-sized coreset has the advantage of enabling us to use static algorithms to solve the clustering problem, using only the points from the coreset. This approach significantly reduces computation time compared to performing the method with all points. If the coreset effectively represents the points, the clusters found will also provide a good solution. When the dataset changes, updating the Cover Tree and finding a new solution with the same approach is expected to be much faster than repeating the static method with all the points, while maintaining good performance in terms of solution quality.

In [6], the authors introduce an augmented version of the Cover Tree with insertion and deletion algorithms. In this work, we have implemented a generalized version of the Cover Tree, which will be formally defined in the following chapters. We present experimental results designed to identify optimal parameters for the data structure and to address potential issues. Subsequently, we utilize the generalized Cover Tree implementation with all points from the input dataset to solve the k-Center problem, both with and without outliers.

In summary, this paper endeavors to explore the potential of Cover Trees in constructing coresets for clustering. We do so by implementing the data structure and the associated algorithms to update the Cover Tree and address the k-Center problem, both with and without outliers. Through a combination of theoretical insights and practical experiments, our objective is to showcase the utility of Cover Trees as a valuable tool for addressing the challenges posed by high-dimensional data in the domain of clustering and data analysis.

1.1 Paper Organization

The upcoming chapters will present both theoretical and practical results, along with considerations regarding our implementations.

The Preliminaries chapter (Chapter 2) provides the foundational definitions and problem statements for our research. It introduces the k-Center problem and its variant with outliers and presents crucial theorems and algorithms that establish the theoretical groundwork.

In the Cover Tree: Data Structure chapter (Chapter 3), we explore the original definition of the Cover Tree as outlined in [5]. This chapter offers historical context and insights into the concept's development. Moving forward, the Generalized Cover Tree chapter (Chapter 4) extends beyond the original Cover Tree definition [6]. Here, we describe a generalized version tailored to our project's specific requirements, bridging the gap between theory and practical applications.

The Solving Clustering Problems chapter (Chapter 5) delves into the algorithms used to address the (Robust) k-Center problem, providing a crucial link between theory and practice. We shift our focus to practical dimensions, presenting empirical results from conducted experiments and offering real-world performance insights in the Experiments chapter (Chapter 6).

Finally, the Conclusions chapter (Chapter 7) summarizes our findings, reflects on research implications, and draws conclusions from both theoretical insights and practical results.

Chapter 2

Preliminaries

Distance metrics, or simply metrics, are mathematical functions that measure the distance between two objects or points. In our work, we have chosen to use the most common distance metric, the Euclidean Distance. The code is developed following a general approach, allowing us to easily switch to another metric if needed.

Definition 2.0.1 (Metric Space)**.** A *metric space* $(X, dist)$ is composed of a set of points X and a distance function $dist : X \times X \to \mathbb{R}$. The following three properties must be satisfied:

1. Non negativity: $dist(x, y) \geq 0$, with $dist(x, x) = 0$
2. Symmetry: $dist(x, y) = dist(y, x)$
3. Triangle inequality: $dist(x, y) \leq dist(x, w) + dist(w, y)$
4. $dist(x, y) = 0$ if and only if $y = x$.

Definition 2.0.2 (Euclidean Distance)**.** The *Euclidean Distance* is the length of a straight line between two points in a physical space, representing the shortest possible path. Given two point with n dimension $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, \cdots, y_n)$, their distance is:

$$dist(X, Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots (x_n - y_n)^2} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

To describe the Cover Tree and establish time complexity metrics, the following definitions will be useful.

Definition 2.0.3 (Ball $\mathrm{B}(p, r)$)**.** Let $(X, dist)$ be a metric space. For any $p \in X$ and $r \geq 0$, the *ball centered at p of radius r*, denoted as $\mathrm{B}(p, r)$ is the subset of all points of X which are at distance of at most r from p.

Definition 2.0.4 (Locally finite space)**.** We say that a subset S of a metric space $(X, dist)$ is *locally finite* if, for all $p \in X$ and $r \in \mathbb{R}_+$, the set $\mathrm{B}(p, r) \cap S$ is finite.

Definition 2.0.5 (Expansion constant)**.** Let S be a locally finite set in a metric space $(X, dist)$. The *expansion constant c* is the smallest value $c \geq 2$ such that $|B(p, 2r)| \leq c \cdot |B(p, r)|$ for any $p \in S$ and $r \geq 0$.

Definition 2.0.6 (Doubling Dimension)**.** The *doubling dimension* of X is the minimum value D such that, for all $p \in X$, any ball $\mathrm{B}(p, r)$ is contained in the union of at most 2^D balls of radius $r/2$ centered at points of X.

The following fact, which is a simple consequence of Definition 2.0.6, was proven in [7].

Fact 1 Let X be a set of points from a metric space of doubling dimension D, and let $Y \subseteq X$ be such that any two distinct points $a, b \in Y$ have pairwise distance $\mathrm{dist}(a,b) > r$. Then for every $R \geq r$ and any point $p \in X$, we have $|\mathrm{B}(p,R) \cap Y)| \leq 2^{\lceil \log_2(2R/r) \cdot D \rceil} \leq (4R/r)^D$. If $R/r = 2^i$, the bound can be improved to $|\mathrm{B}(p,R) \cap Y| \leq (2R/r)^D$.

2.1 k-Center problem

In the realm of data analysis, clustering is a technique used to group similar data points or objects together, or conversely, to separate dissimilar points or objects into distinct groups. Given n data points and a measure of similarity between them, the objective of clustering is to partition the data into k clusters (either predetermined or derived) such that points within the same cluster are considered similar. The choice of clustering problem depends on how we define and measure similarity, as determined by the objective function.

Formally, the k-Center problem is a center-based clustering. The solution to the problem is a set of at most k centers. Each center corresponds to a cluster, and every point is assigned to the cluster with the nearest center.

Definition 2.1.1 (center-based clustering)**.** Let S be a set of n points in the metric space $(X, dist)$ and let $1 \leq k \leq n$ be the target number of clusters. Then we define a k-Clustering of S as a tuple $C = (C_1, C_2, \cdots, C_k; c_1, c_2, \cdots, c_k)$, where $(C_1, C_2, \cdots, C_k)$ defines the partition of S, while $(c_1, c_2, \cdots, c_k)$ are selected centers for the clusters.

Definition 2.1.2 (distance between a point and a set)**.** Let x be a point in the metric space $(X, dist)$ and $S \subseteq X$. Then the distance between x and S is $dist(x, S) = \min_{s \in S} dist(x, s)$.

Definition 2.1.3 (radius)**.** Let $(X, dist)$ be a metric space, and $C \subseteq S \subseteq X$. We define the *radius* of C with respect to S as

$$r_C(S) = \max_{s \in S} dist(s, C)$$

Given C, the optimal clustering is obtained by assigning each point of S to the closest center (with ties broken arbitrarily). We denote the radius of the optimal solution of the k-Center problem as $r^*_k(S)$. In the 80's, Gonzalez [8] developed a 2-approximation sequential algorithm for the k-Center problem that follows a $O(kn)$-time greedy strategy, where n is the number of points to be clustered. In the recent literature, it is referred to as GMM, and it selects the first center arbitrarily as the first center, and then each subsequent center as the point with the maximum distance from the set of previously selected ones. In the same paper, the author also shows that in general metric spaces, it is impossible to achieve an approximation of $2 - \varepsilon$ for any $\varepsilon > 0$, unless $P = NP$.

Algorithm 1 Gonzalez Algorithm

```
Input: A set of point $S$, an integer $k > 0$
Output: A set of Centers of size $k$
1: $C \leftarrow \emptyset$
2: $C \leftarrow C \cup \{s : s \in S\}$
3: while $|C| \neq k$ do
4:     $c = \arg\max_{s \in S} dist(s, C)$
5:     $C \leftarrow C \cup \{c\}$
6: return $C$
```

2.2 Robust k-Center problem

k-Center is useful to guarantee that every point is close to a center. However, for noisy pointsets (e.g., pointsets with outliers), the clustering that optimizes the k-center objective may not distinguish some inherent clustering in the data, as shown in Figure 2.1.

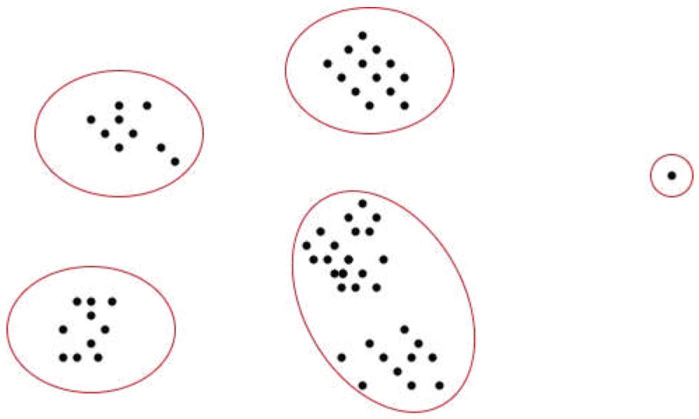


Figure 2.1: 6-Clustering result. Even one outlier changes the "true" clusters.

To deal with noise in the dataset, Charikar et al. [2] introduced the k-center problem with z outliers (also referred to as robust (k, z)-center problem), where the clustering is allowed to ignore z points of the input.

To deal with noise in the dataset, Charikar et al. [2] introduced the k-center problem with z outliers (also referred to as the robust (k, z)-center problem), where the clustering is allowed to ignore z points from the input.

More formally, consider a metric space (X, dist) and a subset $S \subseteq X$ to be clustered. The problem aims to find a subset $C \subseteq S$ of size k that minimizes $r_C(S \setminus Z_C)$, where $Z_C \subseteq S$ is the subset of z points that are at the maximum distance from C, and they are referred to as *outliers*. We denote the radius of the optimal solution of the robust (k, z)-center problem as $r^*_{k,z}(S)$.

Theorem 2.2.1. The optimal solution of the problem without outliers with $k + z$ centers has a smaller radius than the optimal solution of the problem with k centers and z outliers.

$$r^*_{k+z}(S) \leq r^*_{k,z}(S)$$

In [2], the authors present an algorithm that provides a 3-approximation solution running in $O(kn^2 \log n)$ time, where n is the number of points to be clustered. Briefly, given a parameter r, the algorithm constructs a ball of radius r and an expanded ball of radius $3r$ for each point. By definition, every ball of radius r is composed of all the points at a distance less than r. For k iterations, the point c whose ball has the larger size is chosen as a center, and all the points in the expanded ball centered at c are marked as covered points. If the solution has more than z uncovered points, it is considered infeasible. The algorithm performs a binary search to find the minimum r for which the solution is feasible.

Chapter 3

Cover Tree: data structure

Conceptually, a Cover Tree is defined as described in the paper by Beygelzimer et al. [5], specifically in its **implicit representation**. Let S be a set of n points from a metric space (X,dist) that we want to represent. In a Cover Tree T for S, each node corresponds to a point in S, while each point in S is associated with one or more nodes. Each level of the tree is indexed by an integer $\ell \in [-\infty, \infty]$, with integers decreasing as we move downwards from the root towards the leaves. We denote T_ℓ as the set of nodes at level ℓ and $pts(T_\ell)$ as the points associated with the nodes in T_ℓ. For each node $u \in T_\ell$, we maintain its associated point (u.point), a pointer to its parent (u.parent), the list of pointers to its children (u.children), and its level in T (u.level $= \ell$).

For every level ℓ, a Cover Tree must satisfy the following properties:

- **Nesting**: $pts(T_\ell) \subseteq pts(T_{\ell-1})$
- **Covering**: For each $u \in T_\ell$, dist$(u, u.\text{parent}) \leq 2^{\ell+1}$
- **Separation**: For all $u, v \in T_\ell$, dist$(u, v) > 2^\ell$

Definition 3.0.1. Let $d_{\min}$ and $d_{\max}$ be the minimum and the maximum distance between two points in S respectively. The *aspect ratio* Δ of S is defined as $\Delta = d_{\max}/d_{\min}$.

For every $q \in S$, let $\ell(q) = \max\{\ell : q \in pts(T_\ell)\}$. The definition of the tree, specifically the covering condition, implies that for every $\ell \leq \ell(q)$, $q \in pts(T_\ell)$, and the node $u \in T_\ell$ is a *self-child* of itself. This means that u.parent.point$=q$, since for every other node $v \in T_{\ell+1}$ with $v.\text{point} \neq q$, $dist(u, v) > 2^{\ell+1}$.

Since the separation condition holds, for every $\ell < \log_2 d_{\min}$, $pts(T_\ell) = S$. Moreover, due to the covering condition, for every $\ell \geq \log_2 d_{\max}$, $|pts(T_\ell)| = 1$. We can now define:

- $\ell_{\max} = \min\{\ell : |pts(T_\ell)| = 1\}$
- $\ell_{\min} = \max\{\ell : pts(T_\ell) = S\}$

It is also worth noting that every node $u \in T_j$ with $j > \ell_{\max}$ or $j \leq \ell_{\min}$ has only the self child in u.children.

To save space, a more compact representation called **explicit representation** is used [6], where only nodes that have children other than the self-child are maintained. Chains of 1-child nodes, which correspond to instances of the same point, are coalesced. Therefore, the explicit representation considers only the portion of the implicit representation consisting of the levels T_ℓ where $\ell \in [\ell_{\min}, \ell_{\max}]$ and designates the unique node $r \in T_{\ell_{\max}}$ as the root of the tree.

Theorem 3.0.1. The explicit representation of a Cover Tree, which represents a set S of n points, takes $O(n)$ in space, and the number of levels is $O(\log \Delta)$, where Δ is the aspect ratio of S.

Proof. The number of levels is $\ell_{\max} - \ell_{\min}$. Following from the definitions of $\ell_{\max}$ and $\ell_{\min}$, the number of levels is smaller than $\log_2 d_{\max} - \log_2 d_{\min} = \log_2 \Delta$ □

Theorem 3.0.2. Let T be a Cover Tree for S, and consider $u \in T_\ell$. For any descendant u' of u (a child of a child and so on), the following inequality holds:

$$dist(u, u') < 2^{\ell+1}$$

Proof. By the covering condition:

$$dist(u, u') \leq \sum_{i=\ell_{\min}}^{\ell} 2^i < \sum_{i=-\infty}^{\ell} 2^i = 2^{\ell+1}$$

□

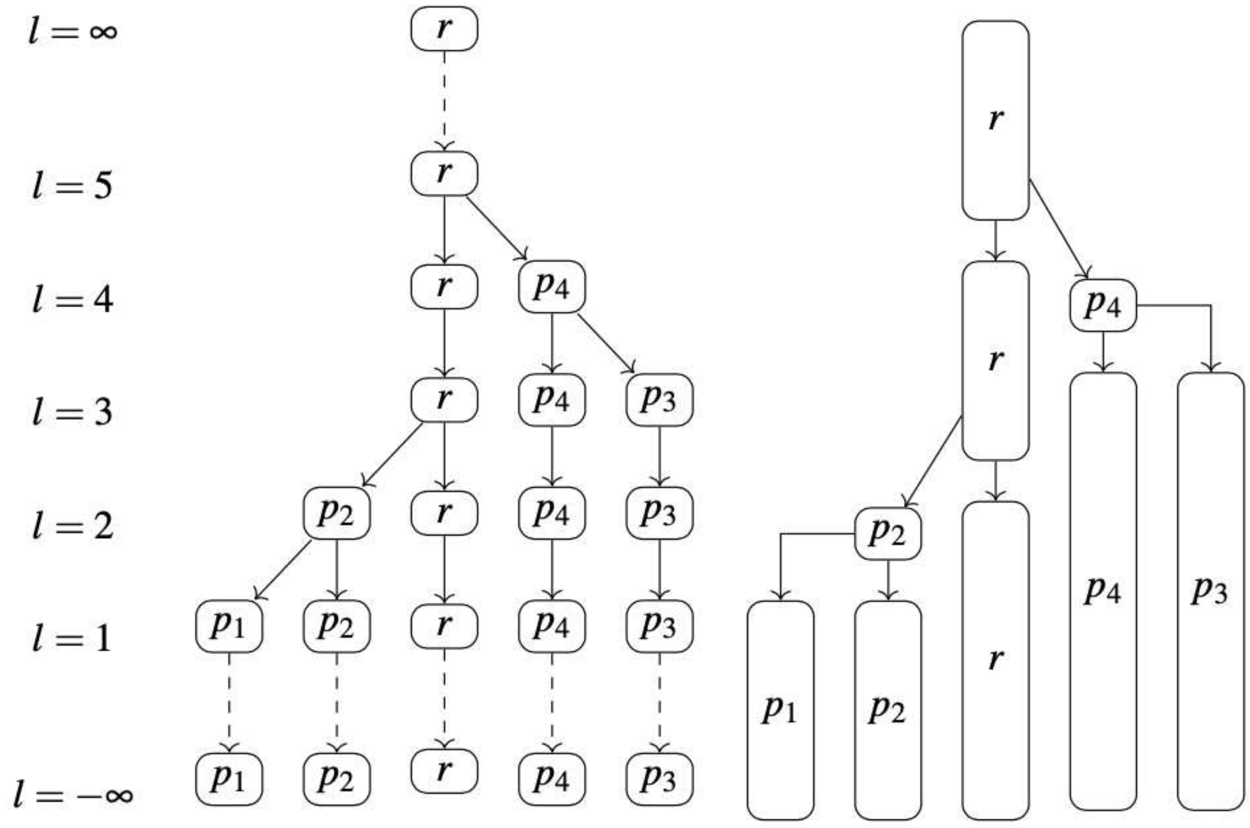


Figure 3.1: Implicit representation on the left and explicit representation on the right of the same Cover Tree [9]

Chapter 4

Generalized Cover Tree

In our implementation, we use the implicit representation of a different type of Cover Tree, named the **Generalized Cover Tree** or **(α, β)-Cover Tree**, which is more suitable for the various datasets of points inserted.

The conditions that each level ℓ must satisfy are as follows:

- **Nesting**: $pts(T_\ell) \subseteq pts(T_{\ell-1})$
- **Covering**: For each $u \in T_\ell$, $\text{dist}(u, u.\text{parent}) \leq \beta \cdot \alpha^{\ell+1}$
- **Separation**: For all $u, v \in T_\ell$, $\text{dist}(u, v) > \beta \cdot \alpha^{\ell}$

This definition was established in "Fully dynamic clustering and diversity maximization in doubling metrics" [6], where the authors provided an augmented definition of the implicit representation of the Cover Tree with fixed $\alpha = 2, \beta = 1$ and presented iterative ascend-descend insertion and deletion algorithms with a doubling dimension-based analysis. In this work, we extend the definition to general α and β.

More specifically, the data structure is augmented by adding an additional field to each node. Given a node $t \in T$, the positive weight $t.\text{weight} = |S_t|$, where S_t is the subset of points of S associated with the nodes in the subtree rooted at t. Therefore, every leaf has a weight of 1, while the root of the tree has a weight of $|S|$.

Adding this additional field doesn't change the space used by the Cover Tree to store all the points, which remains $O(|S|)$. However, it will be used to retrieve a solution for the Robust k-Center problem.

As for the standard Cover Tree (with $\alpha = 2, \beta = 1$), following the same method, it is easy to prove that, for a Cover Tree with a general α, the number of levels is $\log_\alpha \Delta$.

Lemma 4.0.1. Let T be an $(\alpha, 1)$-Cover Tree for S with $\alpha > 1$, and consider $u \in T_\ell$. For any descendant u' of u (a child of a child and so on), the following inequality holds:

$$dist(u, u') < \frac{\alpha^{\ell+1}}{\alpha - 1}$$

Proof. By the covering condition:

$$dist(u, u') \leq \sum_{i=\ell_{\min}}^{\ell} \alpha^i < \sum_{i=-\infty}^{\ell} \alpha^i = \sum_{i=-\infty}^{0} \alpha^i + \sum_{i=1}^{\ell} \alpha^i = \frac{\alpha}{\alpha - 1} + \frac{\alpha^{\ell+1} - \alpha}{\alpha - 1} = \frac{\alpha^{\ell+1}}{\alpha - 1}$$

□

4.1 Insertion

The pseudocode for the insertion algorithm, which inserts a point p into the Cover Tree T, is shown in Algorithm 2. The algorithm begins by descending through the tree to find the smallest level ℓ at which the point p can be inserted among its children while respecting the covering condition (lines 6-14). Afterward, the algorithm ascends to identify the highest level ℓ where the separation condition is guaranteed (lines 15-16).

Once this level is found, p is inserted among the children of a node at level ℓ that adheres to the covering condition. The final while loop is used to update the weights of the nodes that cover the newly inserted point.

When the new point is sufficiently far from the root, $\ell_{\max}$ must be updated to a greater value. This ensures that all the children of the root node, including p, adhere to the covering condition.

Lemma 4.1.1. Let p a point to be insert in an (α, β)-Cover Tree. If $dist(p, r) > \beta \cdot \alpha^{\ell}_{\max}$, then the covering condition is satisfied setting $\ell_{\max} = \lceil \log_\alpha (\text{dist}(p, r)/\beta) \rceil$.

Proof. $\text{dist}(p, r) \leq \beta \cdot \alpha^{\ell_{\max}}$ due to covering condition, so $\alpha^{\ell_{\max}} = \text{dist}(p, r)/\beta$
$\rightarrow \ell_{\max} = \log_\alpha (\text{dist}(p, r)/\beta)$, since the levels are integers, we have to round up the logarithm. □

Algorithm 2 Insertion of a point

```
Input: A Cover Tree with root $r$, a point $p$
 1: if $\text{dist}(p, r) > \beta \cdot \alpha^{\ell_{\max}}$ then
 2:     $\ell_{\max} = \lceil \log_\alpha (\text{dist}(p, r)/\beta) \rceil$
 3:     $r$.level $= \ell_{\max}$
 4: $\ell = \ell_{\max}$
 5: $Q_\ell = \{r\}$
 6: while $Q_\ell \neq \emptyset$ do
 7:     $Q_{\ell-1} = \emptyset$
 8:     for each $q \in Q_\ell$ do
 9:         if $q$.children $== \emptyset$ OR $q$.children[0] $\neq \ell - 1$ then
10:             if $\text{dist}(q, p) \leq \beta \cdot \alpha^\ell$ then
11:                 $Q_{\ell-1} = Q_{\ell-1} \cup \{q\}$
12:         else
13:             $Q_{\ell-1} = Q_{\ell-1} \cup \{q' \in q.\text{children s.t. } \text{dist}(p, q') \leq \beta \cdot \alpha^\ell\}$
14:     $\ell = \ell - 1$
15: while $\text{dist}(p, Q_{\ell+1}) > \beta \cdot \alpha^{\ell+1}$ do
16:     $\ell = \ell + 1$
17: let $v \in Q_{\ell+1}$ be s.t $\text{dist}(p, v) \leq \beta \cdot \alpha^{\ell+1}$
18: $u$ =new node with: $u$.point= $p$ and $u$.level= $\ell$
19: if $v$.children $== \emptyset$ OR $v$.children[0] $\neq \ell$ then
20:     $w$ = new node with: $w$.point = $v$.point, $w$.level = $\ell$
21:     $w$.children = $v$.children and $w$.weight = $v$.weight
22:     $v$.children = $\{w\}$
23:     $v$.children = $v$.children $\cup$ $\{u\}$
24: $t = u$
25: while $t \neq null$ do
26:     $t$.weight++
27:     $t = t$.parent
```

Lemma 4.1.2. Given an $(\alpha, 1)$-Cover Tree, for every point p from (X, dist), $\ell \leq \ell_{\max}$, and $t \in T_\ell \setminus Q^p_\ell$, we have $\text{dist}(p, t) > (\alpha - 1) \cdot \alpha^{\ell+1}$.

Proof. The proof proceeds by (backward) induction on ℓ. For the base case $\ell = \ell_{\max}$, the statement holds vacuously. Assume now that the statement holds at level $\ell+1$, and let $t \in T_\ell \setminus Q^p_\ell$. The statement immediately follows when $t.\text{parent} \in Q^p_{\ell+1}$. Otherwise, since $t.\text{parent} \in T_{\ell+1} \setminus Q^p_{\ell+1}$, by the inductive hypothesis it must be $\text{dist}(p, t.\text{parent}) > \alpha^{\ell+2}$, whence $\text{dist}(p, t) \geq \text{dist}(p, t.\text{parent}) - \text{dist}(t, t.\text{parent}) > \alpha^{\ell+2} - \alpha^{\ell+1} = (\alpha - 1) \cdot \alpha^{\ell+1}$. □

Lemma 4.1.3. Given an $(\alpha, 1)$-Cover Tree, for every point $p \in S$ and $\ell \leq \ell_{\max}$, we have that, when $\alpha \leq 2$:

1. $|Q^p_\ell| \leq 4^D$
2. $\sum_{q \in Q^p_\ell} |q.\text{children}| \leq 16^D$

Otherwise, using a general α:

1. $|Q^p_\ell| \leq (4 \cdot (\alpha^2 - \alpha))^D$
2. $\sum_{q \in Q^p_\ell} |q.\text{children}| \leq (4 \cdot (\alpha^2 + \alpha))^D$

Proof. Consider when $\alpha \leq 2$:

For each $t_1, t_2 \in T_\ell$, we have that $\text{dist}(t_1, t_2) > \alpha^\ell$. Also, $\text{pts}(Q^p_\ell) \subseteq \text{B}(p, (\alpha - 1) \cdot \alpha^{\ell+1}) \cup T_\ell \subseteq \text{B}(p, 2^{\ell+1}) \cup T_\ell$. Then, by Fact 2, it follows that $|Q^p_\ell| \leq (2 \cdot 2^{\ell+1}/2^\ell)^D = 4^D$.

Moreover, for each $q \in Q^p_\ell$ and each $q' \in q$.children, we have that $\text{dist}(q', p) \leq \text{dist}(q', q) + \text{dist}(q, p) \leq \alpha^\ell + \alpha^{\ell+1} = (1 + \alpha) \cdot \alpha^\ell$, whence q'.point $\in \text{B}(p, (1+\alpha) \cdot \alpha^\ell) \cup \text{pts}(T_{\ell-1}) \subseteq \text{B}(p, 4 \cdot 2^\ell) \cup \text{pts}(T_{\ell-1})$. Then, again by Fact 2, it follows that $|\{q' \in q\text{.children s.t. } q \in Q^p_\ell\}| \leq (2 \cdot 4 \cdot 2^\ell/2^{\ell-1})^D = 16^D$.

More generally for every $\alpha > 1$:

$\text{pts}(Q^p_\ell) \subseteq \text{B}(p, (\alpha-1) \cdot \alpha^{\ell+1}) \cup T_\ell$ and by Fact 2, it follows that $|Q^p_\ell| \leq (4 \cdot (\alpha-1) \cdot \alpha^{\ell+1}/\alpha^\ell)^D = (4 \cdot (\alpha^2 - \alpha))^D$.

Moreover, for each $q \in Q^p_\ell$ and each $q' \in q$.children, we have that $\text{dist}(q', p) \leq \text{dist}(q', q) + \text{dist}(q, p) \leq \alpha^\ell + \alpha^{\ell+1} = (1 + \alpha) \cdot \alpha^\ell$, whence q'.point $\in \text{B}(p, (1+\alpha) \cdot \alpha^\ell) \cup \text{pts}(T_{\ell-1})$. Then, again by Fact 2, it follows that $|\{q \in q\text{.children s.t. } q \in Q^p_\ell\}| \leq (4 \cdot (1+\alpha) \cdot \alpha^\ell/\alpha^{\ell-1})^D = (4 \cdot (\alpha^2 + \alpha))^D$. □

Theorem 4.1.1. Let T be an augmented (α, β)-Cover Tree for a set S. The insertion algorithm described above yields an augmented (α, β)-Cover Tree for $S \cup \{p\}$ in time:

- $O(16^D \log_\alpha \Delta)$ when $\alpha \leq 2$
- $O((4 \cdot (\alpha^2 + \alpha))^D \log_\alpha \Delta)$ for every $\alpha > 1$

Where D is the doubling dimension of the metric space and Δ is the aspect ratio of S. The complexity bounds found refer to an $(\alpha, 1)$-Cover Tree.

Proof. The proof follows the same intuition exhibited in Paper [6] using Lemma 4.1.3. □

Note that for $\alpha < (-1 + \sqrt{17})/2 \sim 1.56$, $4 \cdot (\alpha^2 + \alpha) < 16$, so the general bound is tighter. For a smaller α, $\sum_{q \in Q^p_\ell} |q\text{.children}|$ becomes smaller, but $\log_\alpha \Delta$, so the number of levels becomes higher.

4.2 Deletion

The pseudocode for the deletion algorithm, which removes a point p from the Cover Tree T, is shown in Algorithm 3. In the first while loop, the algorithm finds the level ℓ at which the point p, to be deleted, is a leaf node (it has no children).

The algorithm then proceeds to remove p from every level and rebalances the tree. This includes handling cases where the parent of node u with u.point $= p$ has only two children, requiring additional operations to merge the child with the parent.

This procedure continues until the root node is reached. The update of the root is treated separately in lines 40-57.

Algorithm 3 Deletion of a point

```
Input: A Cover Tree with root $r$, a point $p$
 1: $\ell = \ell_{\max}$
 2: $Q_\ell = \{r\}$
 3: while true do
 4:     $Q_{\ell-1} = \emptyset$
 5:     for each $q \in Q_\ell$ do
 6:         if $q == p$ AND $q$.children $== \emptyset$ then
 7:             break while
 8:         if $q$.children $== \emptyset$ OR $q$.children[0] $\neq \ell - 1$ then
 9:             if dist$(q, p) \leq \beta \cdot \alpha^\ell$ then
10:                 $Q_{\ell-1} = Q_{\ell-1} \cup \{q\}$
11:         else
12:             $Q_{\ell-1} = Q_{\ell-1} \cup \{q' \in q$.children s.t. dist$(p, q') \leq \beta \cdot \alpha^\ell\}$
13:     $\ell = \ell - 1$
14: $R_\ell = \emptyset$
15: while $\ell < \ell_{\max}$ do
16:     if $\exists u \in Q_\ell$ s.t. $u$.point $== p$ AND $u$.level $== \ell$ then
17:         $v = u$.parent
18:         delete $u$ from $v$. children
19:         if $v$.point $== p$ then
20:             $R_\ell = R_\ell \cup v$.children
21:         else if $|v$.children$| == 1$ then
22:             $v$.children $= v$.children[0].children
23:             $v$.weight $= v$.children[0].weight
24:             delete $v$.children[0]
25:     $R_{\ell+1} = \emptyset$
26:     for each $w \in R_\ell$ do
27:         if $\exists w' \in Q_{\ell+1} \cup R_{\ell+1}$ s.t. dist$(w, w') \leq \beta \cdot \alpha^{\ell+1}$ then
28:             if $w'$.children $\neq \emptyset$ AND $w'$.children[0].level $== \ell$ then
29:                 $w'$.children $= w'$.children $\cup \{w\}$
30:             else
31:                 $z =$ new node with: $z$.point $= w'$.point, $z$.level $= \ell$
32:                 $z$.children $= w'$.children and $z$.weight $= w'$.weight
33:                 $w'$.children $= \{w, z\}$
34:         else
35:             $w$.level $= \ell + 1$
36:             $R_{\ell+1} = R_{\ell+1} \cup \{w\}$
37:     for each $w \in Q_{\ell+1} \cup R_{\ell+1}$ do
38:         update $w$.weight
39:     $\ell = \ell + 1$
40: if $p \neq r$.point then
41:     if $R_\ell == \emptyset$ then
42:         $\ell_{\max} = r$.children[0].level+1
43:         $r.level = \ell_{\max}$
44:     else
45:         $r_{new} =$ new root with: $r_{new}$.point $= r$.point, $r$.level $= \ell + 1$, $r_{new}$.children $= \{r\} \cup R_\ell$
46:         update $r_{new}$.weight
47: else
48:     if $|R_\ell| == 1$ then
49:         let $v = R_\ell[0]$ be the new root
50:         $\ell_{\max} = r$.children[0].level+1
51:         $v$.level $= \ell_{\max}$
52:         update $v$.weight
53:     else
54:         let $v \in R_\ell$
55:         $r_{new} =$ new root with: $r_{new}$.point $= v$.point, $r$.level $= \ell + 1$, $r_{new}$.children $= R_\ell$
56:         $\ell_{\max} = \ell + 1$
57:         update $r_{new}$.weight
```

Theorem 4.2.1. Let T be an augmented (α, β)-Cover Tree for a set S. The deletion algorithm described above yields an augmented (α, β)-Cover Tree for $S \setminus \{p\}$ in time:

- $O(16^D \log_\alpha \Delta)$ when $\alpha \leq 2$
- $O((4 \cdot (\alpha^2 - \alpha))^{2D} \log_\alpha \Delta)$ for every $\alpha > 1$

Where D is the doubling dimension of the metric space and Δ is the aspect ratio of S. The complexity bounds found refer to an $(\alpha, 1)$-Cover Tree.

Proof. The proof follows the same intuition exhibited in Paper [6] using Lemma 4.1.3. □

In this case for $\alpha = (1 + \sqrt{5})/2 \sim 1.61$, $O((4 \cdot (\alpha^2 - \alpha))^2 < 16$, so the general bound is tighter, but decreasing too much α would increase considerably the number of levels.

Chapter 5

Solving clustering problems

In this chapter, we will explain how to retrieve a solution for the (robust) k-Center problem when all the points are stored in the Cover Tree data structure. We chose to study the Cover Tree data structure because each level of the tree can be seen as a good representation of the lower levels (Figure 5.1).

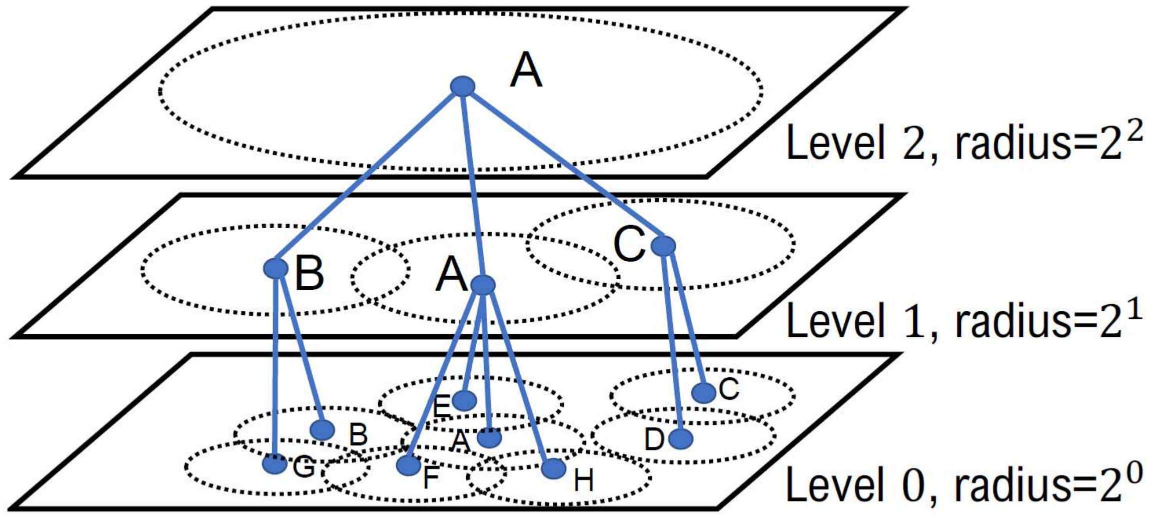


Figure 5.1: Graphical representation of a Cover Tree [10]. In this case the node B in the level 1 can be seen as a representation of G also.

Applying a clustering algorithm to the entire set of points can be computationally expensive. For this reason, we extract a reasonably small representation of the points from the Cover Tree, known as the (ε, k)-coreset.More ε is small, and the higher the quality of the coreset.

Definition 5.0.1 ((ε, k)-coreset)**.** Given a pointset S and a value $\varepsilon > 0$, a subset $R \subseteq S$ is an (ε, k)-coreset for S (w.r.t. the k-center problem) if:

$$r_R(S) \leq \varepsilon r_k^*(S)$$

To extract the coreset from an $(\alpha, 1)$-Cover Tree T, a $(\varepsilon\text{-}k)$ coreset can be obtained by collecting all the points at a particular level ℓ^* that, since the covering condition holds, has all the points in the smaller levels not so far from it. In detail, let $\ell(k)$ be the largest level for which $|T_{\ell(k)}| \leq k$ and $|T_{\ell(k)-1}| > k$. Then:

$$\ell^* = \max\{\ell_{\min}, \ell(k) - \lceil \log_\alpha(2\alpha^2/(\varepsilon(\alpha-1))) \rceil\}$$

$$(\varepsilon\text{-}k) \text{ coreset} = T_{\ell^*}$$

Algorithm 4 Retrieve $(\varepsilon - k)$ coreset

Input: A cover tree T routed in r, an integer $k > 0$, $\varepsilon > 0$
Output: $(\varepsilon - k)$ coreset

```
1: $\ell(k) \leftarrow \ell_{\max}$
2: $T_\ell \leftarrow (\{r\})$
3: while $\ell > \ell_{\min}$ do
4:     $T_{\ell-1} \leftarrow \emptyset$
5:     for each $t \in T_\ell$ do
6:         if $t$.children == $\emptyset$ OR $t$.children[0] $\neq \ell - 1$ then
7:             $T_{\ell-1} \leftarrow T_{\ell-1} \cup \{t\}$
8:         else
9:             $T_{\ell-1} \leftarrow T_{\ell-1} \cup \{t' \in t.\text{children}\}$
10:    if $|T_\ell| \leq k$ then
11:        $\ell(k) \leftarrow \ell$
12:    $\ell \leftarrow \ell - 1$
13: $\ell^* = \max\{\ell_{\min}, \ell(k) - \lceil \log_\alpha(2\alpha^2/(\varepsilon(\alpha-1))) \rceil\}$
14: return $T_{\ell^*}$
```

Lemma 5.0.1. Let T_{ℓ^*} be the level of the augmented $(\alpha, 1)$-Cover Tree returned by the Algorithm 4. The set of points $\text{pts}(T_{\ell^*})$ is an (ε, k)-coreset for S of size at most $k(8 \cdot \alpha^4/(\varepsilon(\alpha-1)^2))^D$ and can be constructed in time $O(k(8 \cdot \alpha^4/(\varepsilon(\alpha-1)^2))^D + log_\alpha \Delta))$.

The proof follows the same intuition exposed in [6], adapting it to a general α instead of 2.

Proof. We first show that $\text{pts}(T_{\ell^*})$ is an (ε, k)-coreset for S. If $\ell^* = \ell_{\min}$, we have $\text{pts}(T_{\ell^*}) = S$, so the property is trivially true. Suppose that $\ell^* = \ell(k) - \lceil \log_\alpha(2\alpha^2/(\varepsilon(\alpha-1))) \rceil > \ell_{\min}$ and consider an arbitrary point $p \in S$. There must exist some node $t \in T_{\ell_{\min}}$ such that $p = t.\text{point}$. Let v be the ancestor of t in T_{ℓ^*}. By Lemma 4.0.1, we know that $\text{dist}(v.\text{point}, p) \leq \alpha^{\ell^*+1}/(\alpha-1) \leq (\varepsilon/2\alpha) \cdot \alpha^{\ell(k)}$. Also, all pairwise distances among points of $\text{pts}(T_{\ell(k)-1})$ are greater than $\alpha^{\ell(k)-1}$. However, since $|\text{pts}(T_{\ell(k)-1})| \geq k+1$, there must be two points $q, q' \in \text{pts}(T_{\ell(k)-1})$ which belong to the same cluster in the optimal k-center clustering of S. Therefore, $\alpha^{\ell(k)-1} < \text{dist}(q, q') \leq 2r_k^*(S)$, which implies that $\alpha^{\ell(k)} \leq 2\alpha \cdot r_k^*(S)$. Putting it all together, we get that for any $p \in S$, $\text{dist}(p, \text{pts}(T_{\ell^*})) \leq \varepsilon r_k^*(S)$.

Let us now bound the size of T_{ℓ^*}. By construction $|T_{\ell(k)}| \leq k$, and we observe that T_{ℓ^*} can be partitioned into $|T_{\ell(k)}|$ subsets $T_u^{\ell^*}$, for every $u \in T_{\ell(k)}$, where $T_u^{\ell^*}$ is the set of descendants of u in T_{ℓ^*}. As stated previously, for each $u \in T_{\ell(k)}$ and $v \in T_u^{\ell^*}$, $\text{dist}(u, v) \leq \alpha^{\ell(k)+1}/(\alpha-1)$.

Moreover, since the pairwise distance between points of $T_u^{\ell^*}$ is greater than α^{ℓ^*}. By applying Fact 2 with $Y = T_u^{\ell^*}$, $R = \alpha^{\ell(k)+1}/(\alpha-1)$, and $r = \alpha^{\ell^*}$, we obtain that

$$|T_u^{\ell^*}| \leq (4 \cdot (1/(\alpha-1))\alpha^{\ell(k)+1-\ell(k)+\lceil \log_\alpha(2\alpha^2/\varepsilon(\alpha-1))\rceil})^D \leq (8 \cdot \alpha^4/(\varepsilon(\alpha-1)^2))^D$$

and the bound on $|T_{\ell^*}|$ follows. T_{ℓ^*} can be constructed on the explicit tree through a simple level-by-level visit up to level ℓ^*, which can be easily determined from $\ell(k)$ and the fact that $\ell_{\min}$ is the largest level ℓ for which all nodes in T_ℓ only have the self-child. The construction time is linear in

$$\sum_{\ell=\ell^*}^{\ell_{\max}} |T_\ell| = \sum_{\ell=\ell^*}^{\ell(k)-1} |T_\ell| + \sum_{\ell=\ell(k)}^{\ell_{\max}} |T_\ell|$$

The second summation is clearly upper-bounded by $k \log_\alpha \Delta$, while, using again Fact 2, it is easy to argue that $|T_\ell| \leq k \cdot 4 \cdot \alpha^{(\ell(k)+1-\ell)} \cdot D$, for every $\ell^* \leq \ell \leq \ell(k)-1$, whence the first sum is $O\left(k(8 \cdot \alpha^4/(\varepsilon(\alpha-1)^2))^D\right)$. The lemma follows.

□

The above proof shows that for every point $t \in T_{\ell^*}$ and for any point point d in the descendants of t in T, $dist(t,d) \leq \varepsilon r_k^*(S)$.

5.1 k-Center

In this section we will explore a method for the k-Center problem that gives better theoretical time complexity than the Gonzalez Algorithm 1.

To retrieve a solution for the k-Center problem, we compute an (ε-k) coreset as explained below, and then run the sequential algorithm 5 using pts(T_{ℓ^*}) as the input set of points.

Given a hyperparameter m, once the coreset is computed, we construct m generalized Cover Trees for the coreset, setting $\beta = \alpha^{p/m}$ with $1 \leq p \leq m$.

The solution to the problem is the set of points pts($T^p_{\ell(k)_p}$) of Cover Tree p that minimizes $\alpha^{\ell(k)_p + p/m}$.

Algorithm 5 Solve K-Center Algorithm

```
Input: A Cover Tree $T$, an integer $k > 0$
Output: A set of Centers of size $k$
1: $Q \leftarrow$ ($\varepsilon$-$k$) coreset($T$)
2: for $1 \leq p \leq m$ do
3:     $T_p$ = ($\alpha$-$\alpha^{p/m}$) Cover Tree ($Q$)
4:     $\ell(k)_p$ = minimum level s.t. pts$|T_{\ell(k)_p}| \leq k$
5: $p^* = \arg\min_{p \in [1,m]} \alpha^{\ell(k)_p + p/m}$
6: $C \leftarrow$ pts ($T^{p^*}_{\ell(k)_{p^*}}$)
7: return $C$
```

In [6] it has been proven that the algorithm has a linear dependence on k in terms of computational time.

Theorem 5.1.1. Given an augmented $(\alpha, 1)$, Cover Tree T for S, by selecting $m = O\left(\frac{\ln(1/\varepsilon)}{\varepsilon}\right)$, the above procedure returns a $(2 + O(\epsilon))$-approximation C to the k-center problem for S, and can be implemented in time $O\left(\frac{k}{\epsilon}\left(\frac{32\cdot\alpha^5(\alpha+1)}{\epsilon(\alpha-1)^2}\right)^D \log_\alpha \Delta\right)$.

Proof. The proof that the above procedure returns a $(2 + O(\epsilon))$-approximation C to the k-center problem for S was proven in [6] and it is independent from α. As for the running time, the construction of the (ϵ, k)-coreset Q requires $O(k((8 \cdot \alpha^4/\varepsilon)^D + log_\alpha\Delta))$ time (shown in Lemma 5.0.1).

The running time of Phase 2 is dominated by the construction of the $m = O\left(\frac{\ln(1/\varepsilon)}{\varepsilon}\right)$ $(\alpha, \alpha^{p/m})$-cover trees $T^{(p)}$, for $1 \leq p \leq m$, achieved through successive insertions of the elements of Q. As previously observed, each insertion takes $O((4 \cdot (\alpha^2 + \alpha))^D \log_\alpha \Delta)$ time. Therefore, the cost of constructing each $T^{(p)}$ is $O\left(|Q|(4 \cdot (\alpha^2 + \alpha))^D \log_\alpha \Delta\right)$. Since $|Q| \leq k(8 \cdot \alpha^4/(\varepsilon(\alpha - 1)^2))^D$, the total cost is thus $O\left(\frac{k}{\epsilon}\left(\frac{32\cdot\alpha^5(\alpha+1)}{\epsilon(\alpha-1)^2}\right)^D \log_\alpha \Delta\right)$, which dominates over the cost of Phase 1.

□

If we use the same approach, changing only the sequential algorithm, Gonzalez has a quadratic dependence on k because it also depends on the size of the input. This leads to a time complexity to compute the solution using a retrieved coreset of $O\left(k^2\left(\frac{8\cdot\alpha^4}{\varepsilon(\alpha-1)^2}\right)^D\right)$. Despite this, as will be shown in Section 6, once the coreset is computed, for reasonable k, it is more convenient to use the Gonzalez algorithm since inserting a point into a Cover Tree is rather time-expensive.

5.2 k-Center with z outliers

Similar to the approach to solve the k-Center problem, first, we have to extract an $(\varepsilon\text{-}k + z)$ coreset, named Q. Each point $q \in Q$ has a weight derived from the node in the generalized cover tree from which q was extracted.

Lemma 5.2.1. Let $r^*_{k+z}(S)$ be the objective function of the optimal solution of the $(k + z)$ center problem and $r^*_{k,z}(S)$ for the k-Center problem with z outliers, then it is straightforward that:

$$r^*_{k+z}(S) \leq r^*_{k,z}(S)$$

Based on this lemma, let Q be the derived $(\varepsilon\text{-}k + z)$ coreset, then $r_Q(S) \leq \varepsilon r^*_{k+z}(S) \leq \varepsilon r^*_{k,z}(S)$.

Once we extract the weighted coreset, we repeatedly utilize the OutliersCluster algorithm (Algorithm 6), as explained in [11]. OutliersCluster(S, k, r, ε) is a modification of the Charikar Algorithm [2], introducing an approximation parameter ε. The algorithm starts by marking all the input points S as uncovered. Then, at each iteration, it chooses the center c that maximizes the number of points in the ball centered at c with a radius of $(1 + 2\varepsilon) \cdot r$ and removes all the points in the expanded ball centered at c with a radius of $(3 + 4\varepsilon) \cdot r$ from the set of uncovered points. This algorithm returns a set $C \subseteq S$ of at most k centers and a set $S' \subseteq S$ of uncovered points when it selects k centers or covers all the points.

To determine the final solution of the problem, we continue running OutliersCluster until the set S' has an aggregate weight $W = \sum_{s \in S'} w_s \leq z$. In particular, we set $r = \frac{\alpha^{\ell_{\max}}}{(1+\varepsilon)^i}$, with i being an integer that increases from 0 in every iteration of OutliersCluster, and then return the set C as the final solution when $W \leq z$ and r is as small as possible. If $\varepsilon > 0$, as i increases, the radius r decreases. Consequently, the selected center will have an expanded ball with a smaller radius, covering fewer points. This, in turn, leads to an increase in the aggregate weight W of the set of uncovered points.

Theorem 5.2.1. Given an augmented $(\alpha, 1)$-Cover Tree T for S, the above procedure returns a $(3 + O(\varepsilon))$-approximation C to the k-center problem with z outliers for S, and can be implemented in time $O\left(\frac{(k+z)^2}{\varepsilon}\left(\frac{8 \cdot \alpha^4}{\varepsilon(\alpha-1)^2}\right)^{2D} \log \Delta\right)$.

Proof. The proof is equal to the one in [6] since we use the same cluster to the same algorithm OutliersCluster. For what concern the time complexity, the running time is dominated by the repeated executions of OutliersCluster.

Given a coreset Q as input, each execution of OutliersCluster can be performed in $O\left(|Q|^2 + k|Q|\right) = O\left((k+z)^2(8 \cdot \alpha^4/(\varepsilon(\alpha-1)^2))^{2D} + k \cdot (k+z)(8 \cdot \alpha^4/(\varepsilon(\alpha-1)^2))^D\right) = O\left((k+z)^2(8 \cdot \alpha^4/(\varepsilon(\alpha-1)^2))^{2D}\right)$ time. The bound on the running time follows by observing that $\alpha^{\ell_{\max}}/r^*_{k,z}(S) = O(\Delta)$, whence $O\left(\log_{1+\epsilon} \Delta\right) = O\left(\left(\frac{1}{\epsilon}\right) \log \Delta\right)$ executions suffice.

□

Algorithm 6 OutliersCluster

Input: A set of points S, an integer $k > 0$, $r, \hat{\varepsilon} > 0$
Output: A set of Centers C of size k and a set of outliers S'

```
1: $S' \leftarrow S$
2: $C \leftarrow \emptyset$
3: while $|C| < k$ AND $S' \neq \emptyset$ do
4:     for $s \in S$ do
5:         $B_s \leftarrow \{v : v \in S' \wedge d(v,s) \leq (1 + 2\hat{\varepsilon}) \cdot r\}$
6:     $c \leftarrow \arg\max_{s \in S} \sum_{v \in B_s} w_v$
7:     $C \leftarrow C \cup \{c\}$
8:     $E_c \leftarrow \{v : v \in S' \wedge d(v,c) \leq (3 + 4\hat{\varepsilon}) \cdot r\}$
9:     $S' \leftarrow S' \setminus E_c$
10: return $C, S'$
```

Chapter 6

Experiments

To test the designed algorithms and the utility of the Cover Tree data structure, we conducted a suite of experiments aimed at measuring the performance of the insertion and deletion algorithms, as well as evaluating the quality of the coresets retrieved from the tree for use in clustering methods. The entire code was implemented in *C++*.

6.1 Machine Settings

All the data presented in this section were collected using a machine with the following technical specifications:

- CPU Intel® i7-1165G7 4,7 GHz quad-core
- RAM 16 GB
- C++ 20

For plotting the data, comparing different algorithms, and tuning some parameters, we used Microsoft Excel 2023.

6.2 Datasets

As we will demonstrate below, inserting (or deleting) a point into (from) the Cover Tree is computationally expensive in terms of time consumption. For this reason, we used datasets with around 2 million points.

- *HAR70+* (Human Activity Recognition) [12] contains 2,259,597 points. This dataset consists of 18 fit-to-frail older-adult subjects (70-95 years old) wearing two 3-axial accelerometers, attached to the right thigh and lower back, for approximately 40 minutes during a semi-structured free-living protocol. Each point has 6 dimensions.

- *Power* [13] comprises 2,075,259 points, which are measurements of electric power consumption in a single household located in Sceaux (7km from Paris, France). The data has a one-minute sampling rate over a period of almost 4 years (between December 2006 and November 2010). Each point has 7 dimensions.

- *MetroPT-3* [14] includes 1,516,948 points representing readings from pressure, temperature, motor current, and air intake valves collected from a compressor's Air Production Unit (APU) on a metro train in an operational context. This data was collected to support the development of predictive maintenance, anomaly detection, and remaining useful life (RUL) prediction models for compressors using deep learning and machine learning methods. Each point has 15 dimensions, but we decided to use only 7 of them since the others are 0-1 values (as labels) or indexes.

6.3 Insertion Testing

As mentioned earlier, the insertion algorithm boasts good theoretical time complexity bounds, but in practice, its performance is poor. After testing the algorithm on a very small dataset (Iris [15]) and confirming that it constructs a Cover Tree that adheres to the three constraints, the next step was to measure the time consumed when dealing with a much larger dataset, namely the *Power* dataset.

Inserting the entire 2 million points using the standard Cover Tree (with $\alpha = 2$ and $\beta = 1$) resulted in an unreasonable time of approximately 1 hour. To address this issue, in order to avoid expensive square-root computations, we decided to use as distance the square of the Euclidean distance, we experimented with different values of α (the standard Cover Tree would have $\alpha = 4$ and $\beta = 1$) to check whether better running times could be obtained.

Here's how we proceeded: we sampled 50,000 points from the dataset and adjusted α, also measuring the average degree of the nodes in the tree while excluding the leaves (since including them would lead to an overall average degree of $(n-1)/n$ as in all trees). Additionally, we counted the levels of the tree crossed.

The distances present in the generalized Cover Tree definition are compared to $\beta \cdot \alpha^{\ell}$. The relationship between the examined level ℓ and α is exponential, while β is used as a multiplicative factor, since that, we set $\beta = 1$ as in the standard Cover Tree and only modify the α parameter. Once we determine the most suitable α for all three datasets, we collect statistics for smaller and larger α values to highlight how the statistics change when α is modified.

The data in the following tables demonstrate that the algorithm is sensitive to the degree of the nodes, its relationship with α, is directly proportional. However, the degree is inversely proportional to the levels crossed.

POWER $\sim$ 2M Points	$\alpha = 1.3$	$\alpha = 1.7$	$\alpha = 2.5$
Average Degree	3.12	3.25	3.47
Average Levels Walk Through	58.4	33.1	19.1
Time Consumed (Seconds)	990	720	1057

Table 6.1: Statistics when all the points of Power are inserted in the Cover Tree, with different α.

Clearly, the algorithm performs better when the degree and the levels crossed are small. However, since they are inversely proportional, we need to find a good trade-off between the two. It is important to emphasize that there is no fixed combination of α and β that performs well for all datasets, as the objective was to find the best possible parameters for each set of input points. As seen in the tables, the best parameters found are $\alpha = 1.7$ for Power, $\alpha = 2$ for Metro, and $\alpha = 1.4$ for HAR70+.

METRO $\sim$ 1.52M Points	$\alpha = 1.5$	$\alpha = 2$	$\alpha = 3$
Average Degree	3.20	3.39	3.71
Average Levels Walk Through	40.5	24.9	16.8
Time Consumed (Seconds)	413	371	537

HAR70+ $\sim$ 2.26M Points	$\alpha = 1.1$	$\alpha = 1.4$	$\alpha = 2$
Average Degree	3.08	3.32	3.78
Average Levels Walk Through	118.9	36.7	20.2
Time Consumed (Seconds)	3011	1550	2971

Table 6.2: Statistics when all the points of Metro and Har70+ are inserted in the Cover Trees, with different α.

To find the best α, we initially sampled $50,000$ points and adjusted α to optimize the algorithm's speed. Then, we repeated the same procedure with $500,000$ points, using the best candidates for α identified in the previous step. As mentioned earlier, these parameters vary as the set of input points changes, but what is the relationship between them?

Since the parameters are only used in comparisons with the distance between two points, we attempted to answer this question by measuring the average distance between two points in the dataset, this experiment will be explained in a following Section 6.5.

6.3.1 Profiling of the code

Once the parameters have been set, the time required for insertion drops considerably, but it still remains significant when compared to the time required for queries made to it. Given this, we conducted experiments to identify weaknesses in the code.

Our first attempt was to understand how the size of the Cover Tree influences the insertion of a point into it. To do so, we inserted $100,000$ points at a time and measured the time it took to insert that section of the input set into the data structure.

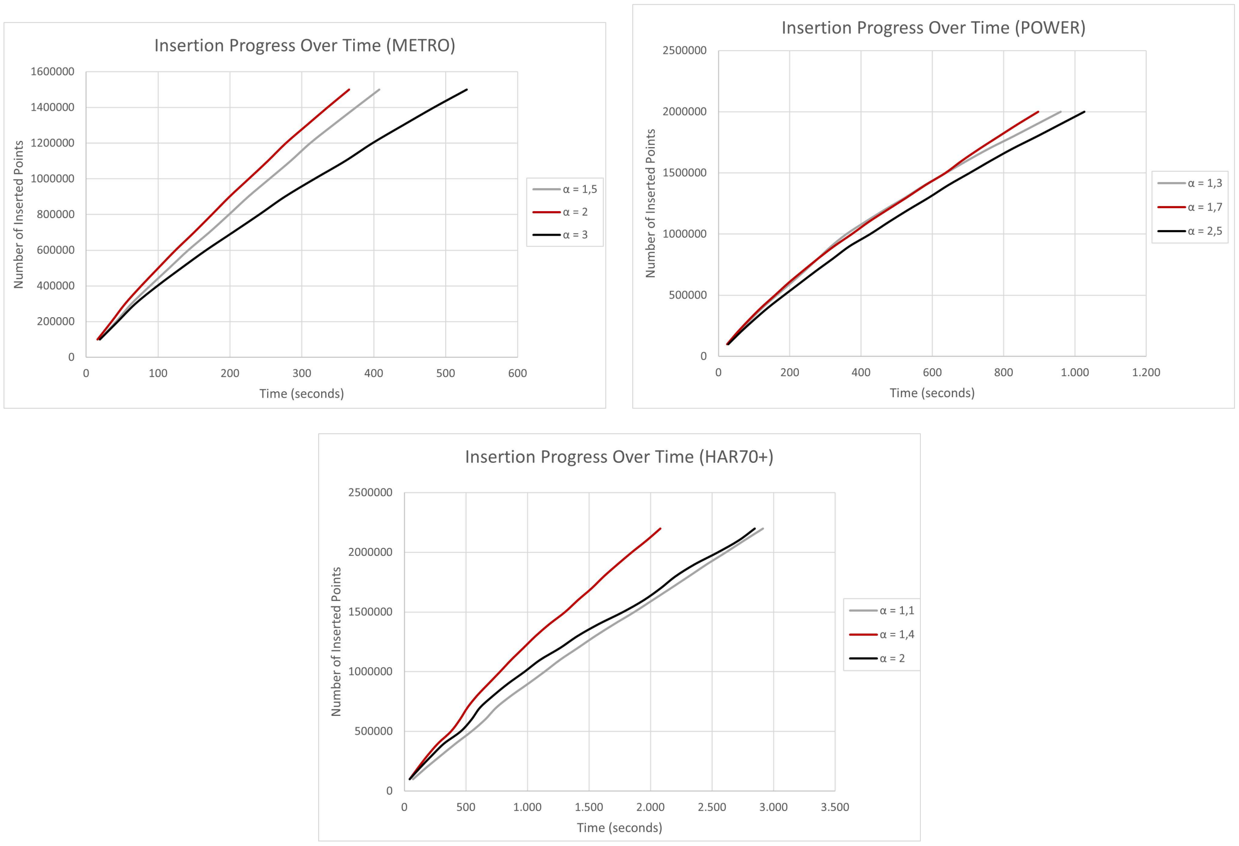
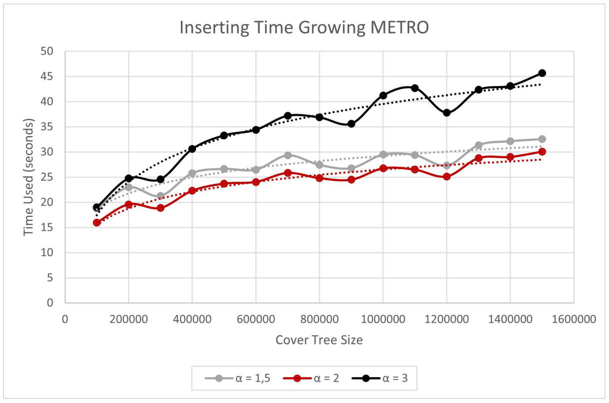


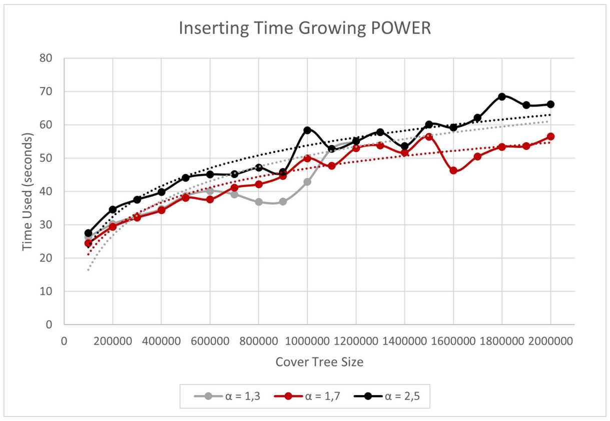
Figure 6.1: These graphs illustrate the number of data points inserted into the data structure over a specified time period with 3 different α for all the datasets.

As expected, the time required to insert a point increases as the size of the data structure grows. However, except for the difference between the first $100,000$ points and the second set of $100,000$, the increase in time is not substantial, demonstrating a logarithmic dependence.

The graphs in Figure 6.1 illustrate the number of points inserted into the data structure over time and how different values of α influence the insertion time for all the points. When comparing different datasets in input, it becomes evident that as the computational time increases, the ratio between the times using different α values also grows.

After examining the time-lapse of the insertion algorithm, our next objective was to determine how the time required to insert 100,000 points (equivalent to 5% of 2 million) increases as the size of the tree expands.





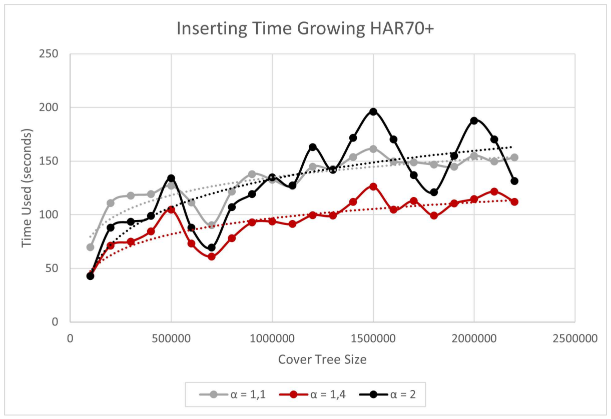


Figure 6.2: These graphs illustrate the time used by the insertion algorithm to insert 100K data points with different sizes of the Cover Tree, with 3 different α for all the datasets.

For the Metro and Power datasets, the increase in insertion time fits well to a logarithmic regression model. However, for HAR70+, the measured time doesn't conform to any specific regression pattern. Overall, it increases slowly but it is highly affected by noise.

Our analysis demonstrates that the size of the tree does influence the insertion time with a logarithmic dependence. The randomness observed in the HAR70+ dataset seems to be due to machine congestion during heavier insertions. On average, the algorithm takes four times as long as for Power and five times as long as for Metro.

Having established that the size of the data structure is not a significant problem, we proceeded to identify which part of the code is the slowest. We partitioned the code into five sections, as referenced in Pseudocode 2:

- The first "if" statement is used to measure the impact of the logarithm computation [lines 1-3].
- The second part involves the first "while" loop [lines 6-14], where all the CoverSets are computed until the point we want to insert is still covered. We aim to capture the impact of this section.
- The third part focuses on the "second while" loop [lines 15-16], where we find the highest level at which the separation constraint is met.
- The fourth part encompasses lines 17-23, where the point is actually inserted into the tree.
- The final part pertains to the "last while" loop [lines 25-27], where the weight of the nodes is updated.

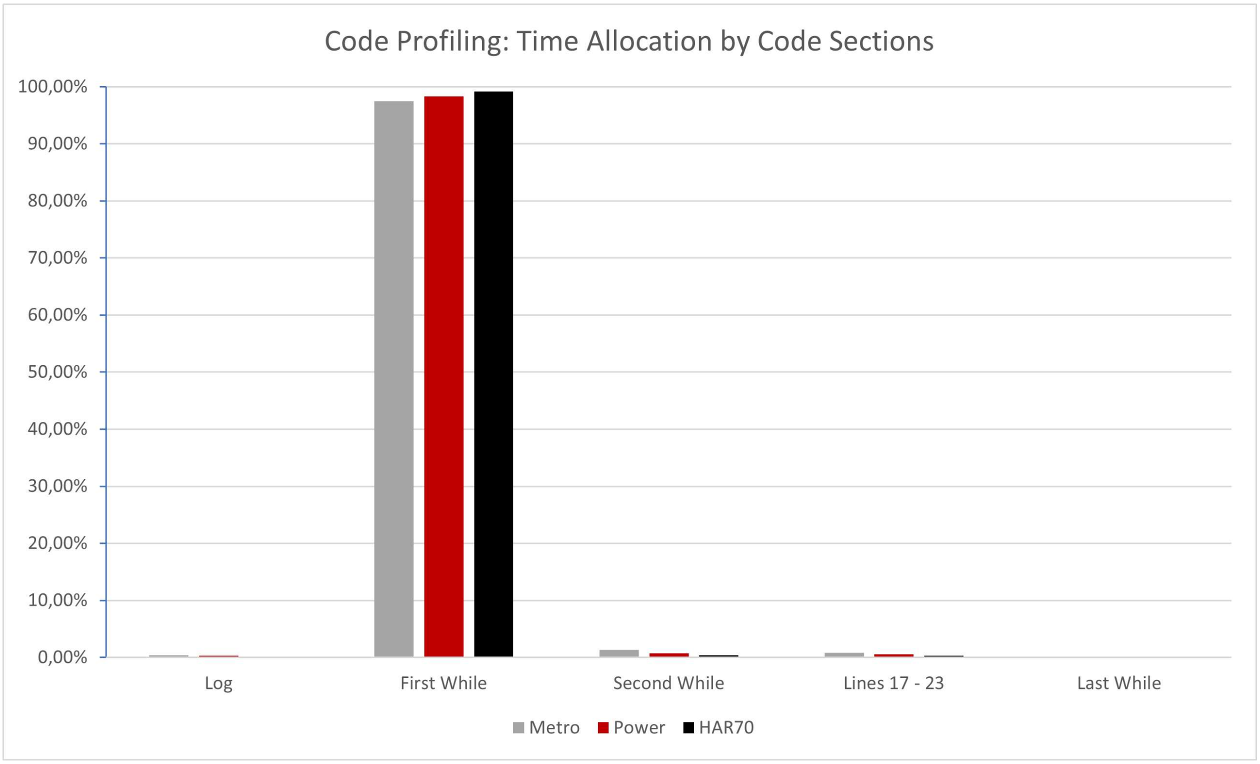


Figure 6.3: Percentage of the time used by different sections of the code.

The results reveal that in all datasets, the part of the code that consumes the majority of the time is the "first while" loop, accounting for over 95% of the total execution time. This outcome is expected since this section is responsible for creating all the CoverSets, and in doing so, it traverses all the nodes in each CoverSet.

Consequently, the average degree of every node emerges as an important statistic. When a CoverSet $Q_{\ell-1}$ is created, for each node in Q_ℓ that has children in $\ell - 1$, the algorithm computes the distance between the point to be inserted and its children.

Figure 6.4 highlights the differing impact of the less time-consuming sections, which can be challenging to discern in Figure 6.3 due to the substantial disparity in time consumed by the "first while" loop compared to the other sections.

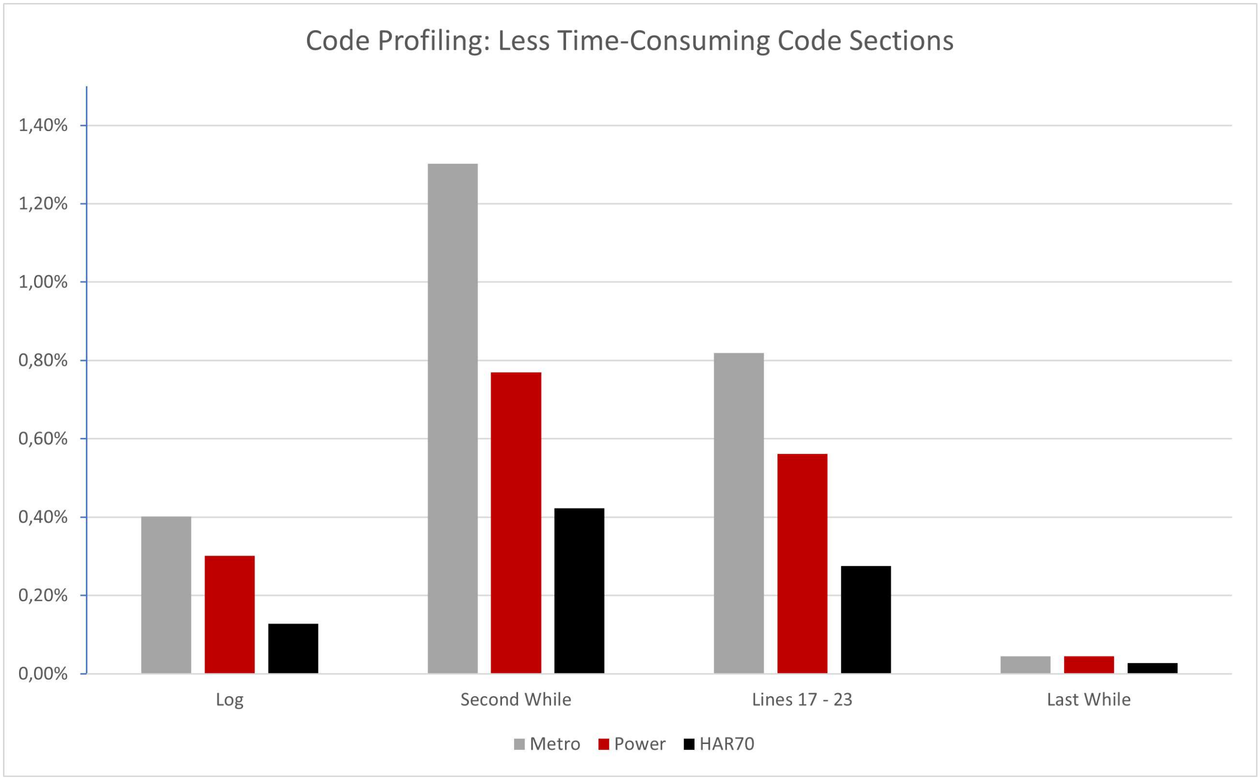


Figure 6.4: Zoom on the less time-consuming sections, all the sections require less than 2% of time.

Across all sections, the time required is less than 2%. However, for smaller datasets or, more generally, inputs where the insertion time is more pronounced, the time ratio between the different sections of the code becomes more distinct.

In the next step, after identifying the most expensive part of the code, we aimed to gather statistics about how the size of the CoverSets changes when varying α. Specifically, we collected the sum of the sizes of the CoverSets and the percentage of discarded points when a comparison is made.

METRO $\sim$ 1.52M Points	$\alpha = 1.5$	$\alpha = 2$	$\alpha = 3$
Average Cardinality CoverSets	113.3	76.9	75.8
Average discarded Points	32.98	47.21	61.45
Time Consumed (Seconds)	413	371	537

Power $\sim$ 2M Points	$\alpha = 1.3$	$\alpha = 1.7$	$\alpha = 2.5$
Average Cardinality CoverSets	181.0	103.8	96.7
Average discarded Points	28.0	45.2	58.9
Time Consumed (Seconds)	990	720	1057

HAR70+ $\sim$ 2.26M Points	$\alpha = 1.1$	$\alpha = 1.4$	$\alpha = 2$
Average Cardinality CoverSets	751.6	222.4	187.8
Average discarded Points	24.6	49.9	64.8
Time Consumed (Seconds)	3011	1550	2971

Table 6.3: Statistics when all the points of Metro, Power and Har70+ are inserted in the Cover Trees, with different α.

Across all combinations of datasets and α, it's evident that a higher percentage of discarded points corresponds to a lower average size of the CoverSets. Once again, it appears that the degree of the nodes plays a critical role in insertion time. When α increases, the only parameter that deteriorates (increases) is the average degree of the nodes. This highlights the sensitivity of the code to the loop where it checks the children of the nodes in the CoverSets, as indicated in line 13 of Algorithm 2.

It's crucial to emphasize that we cannot simply reduce the degree of the nodes by decreasing α, as doing so would negatively impact all the other statistics. It's a trade-off to construct a reasonably balanced tree.

6.4 Comparison between insertion and deletion

Repeating the same analysis also for deletion would be much more difficult. The deletion algorithm is much more complex, as it requires inputting a point that is already in the tree. To conduct similar tests for deletion, we would need to erase the entire tree The problem is that deleting is much more sensible to the permutation of the points in input (deleting a point that is present only in a leaf node or deleting the root has not the same effort).

To assess the relative complexity of insertion and deletion, we opted for a comparative approach. We inserted one million points into the tree (repeated the experiment for each dataset) and then inserted and deleted one point, measuring the time for each operation and repeating it 1000 times. This approach allowed us to gather statistics for both insertion and deletion algorithms.

To test the deletion algorithm, we reversed the process, starting with a tree containing one million points. We then deleted a point from the tree, measured the time for each deletion and reinserted the point back into the tree, repeating the process 1000 times. This ensures that all 1000 insertions and deletions occur with the same Cover Tree, which contains one million points.

The results, as shown in Table 6.4, confirm our thesis. The deletion algorithm consistently proves to be slower, with a larger variance, across all input datasets and for every value of α. An exception is observed for HAR70+ with $\alpha = \{1.4, 2\}$, where deletion times increase differently from insertion times and exhibit lower variance. These tables also highlight the influence of the α parameter on the deletion algorithm, emphasizing the crucial role of the tree structure in both insertion and deletion operations.

METRO $\sim$ 1.52M Points	$\alpha = 1.5$	$\alpha = 2$	$\alpha = 3$
1000 Points Insertion Time (Seconds)	0.27434	0.25468	0.45279
1000 Points Deletion Time (Seconds)	0.43614	0.35721	0.51273
Average Insertion Time (Seconds)	0.00027	0.00025	0.00045
Average Deletion Time (Seconds)	0.00044	0.00036	0.00051
Variance Insertion Time	4.025e-09	3.266e-09	4.224e-07
Variance Deletion Time	1.904e-08	1.029e-08	2.902e-08

Power $\sim$ 2M Points	$\alpha = 1.3$	$\alpha = 1.7$	$\alpha = 2.5$
1000 Points Insertion Time (Seconds)	0.45846	0.37460	0.67990
1000 Points Deletion Time (Seconds)	0.56361	0.42811	0.70015
Average Insertion Time (Seconds)	0.00046	0.00037	0.00068
Average Deletion Time (Seconds)	0.00056	0.00043	0.00070
Variance Insertion Time	8.082e-09	5.310e-09	4.924e-08
Variance Deletion Time	3.356e-08	1.504e-08	5.636e-08

HAR70+ $\sim$ 2.26M Points	$\alpha = 1.1$	$\alpha = 1.4$	$\alpha = 2$
1000 Points Insertion Time (Seconds)	1.22559	0.76105	1.19693
1000 Points Deletion Time (Seconds)	2.00629	0.94395	1.38386
Average Insertion Time (Seconds)	0.001226	0.00076	0.00120
Average Deletion Time (Seconds)	0.00201	0.00094	0.00138
Variance Insertion Time	2.167e-07	1.891e-07	7.264e-07
Variance Deletion Time	1.966e-07	9.240e-08	6.112e-07

Table 6.4: Measured time when we insert 1000 random points from the datasets or when we delete 1000 random points from the tree, with different α. The cover Tree has 1M points inserted in before the insertions and the deletions.

More precisely, on average, the deletion algorithm takes, on average, 37% more time than the insertion algorithm for METRO, with a variance that is 165% greater. For POWER, it takes, on average, 13% more time and results in 171% more variability. Finally, for HAR70+, the deletion of a point takes 34% more time than the insertion of a point, but it results in less variability (25% less variable than the insertion).

Overall, if we consider all the runs, the deletion algorithm is, on average, 28% slower than the insertion, with a variance that is 104% higher.

6.5 Tuning of alpha

As we stated before, we proved in the previous experiments that tuning the α parameter gains considerable benefits in terms of performance for all the datasets.

In particular to capture the dependency between the correct α and the distances between points, we calculate the average distance between two points of the datasets used. Since computing all the distances between all the possible pairs of points from the datasets would lead to a considerable amount of time, we decided to sample $50K$ points for every dataset. We derived also the average norm of the sampled points.

As previous the distances are calculated using the square of the euclidean distance for optimization purpose, while the norm was calculate following the vector norm definition.

	Metro	Power	HAR70+
Number of points	1516948	2049280	2259597
Average Distance	140	342	1.85
Average Norm	64.6	241	1.48
Best α found	2	1.7	1.4

Table 6.5: Statistics of the input datasets.

Metro and Power yield results with statistics of the same order of magnitude, while HAR70+ contains points that are very close to each other, with a norm near 1. Since the best α for Metro is greater than that for Power, we cannot assume that closer points require a smaller α. Also, the number of points doesn't play an important role, as in Section 6.3.1, we demonstrated that the computation time's growth has a logarithmic dependence on the size of the tree (the number of inserted points).

It is important to highlight that the difference in α of 0.3 between Power and Metro, and Power and HAR70+, doesn't influence the results in the same way. In fact, when α is close to 1, even small perturbations significantly affect the data structure and, consequently, the computation time, as shown in previous sections. We can conclude that Metro and Power are similar datasets, not only in terms of average distance and average norm but also in terms of the choice of α.

Thus, we cannot assume that closer points correspond to a smaller α; probably, the choice of a suitable α depends on the simplicity of the datasets. This feature can be captured by the doubling dimension, which is not directly correlated with the distances between the points. Therefore, a simpler dataset requires a greater α, and according to the results of our experiments, this corresponds to lower required computation time. However, capturing the doubling dimension is a challenging task.

6.6 Clustering Algorithms

The environment used for all the clustering experiment is a Cover Tree with all the inserted points, then we will use different clustering algorithms, measuring performances in terms of time and objective function. All the algorithms are independent from k, z and ε, so we used $k = \{10, 50, 100\}$ for k-Center, and $z = 20, 100$ for k-center with z outliers. For what concern setting ε, we decided to make the code completely independent from it, the computation of the coreset is modified as in Algorithm 7, changing the parameter μ to increment the size of the coresets. In this way we can retrieve coresets Q_μ with $|Q_{\mu_1}| \leq |Q_{\mu_2}|$ if $\mu_1 \leq \mu_2$. The values used are $\mu = \{1, 2, 4, 8, 16\}$.

Algorithm 7 Retrieve $(\mu \cdot k)$ coreset

```
Input: A cover tree $T$ routed in $r$, integers $k > 0$ $\mu > 0$
Output: coreset of size $\mu \cdot k$
 1: $\ell \leftarrow \ell_{\max}$
 2: $T_\ell \leftarrow \{r\}$
 3: while $\ell > \ell_{\min}$ do
 4:     $T_{\ell-1} \leftarrow \emptyset$
 5:     for each $t \in T_\ell$ do
 6:         if $t$.children == $\emptyset$ OR $t$.children[0] $\neq \ell - 1$ then
 7:             $T_{\ell-1} \leftarrow T_{\ell-1} \cup \{t\}$
 8:         else
 9:             $T_{\ell-1} \leftarrow T_{\ell-1} \cup \{t' \in t$.children $\}$
10:     $\ell \leftarrow \ell - 1$
11:     if $|T_\ell| \geq \mu \cdot k$ then
12:         break
13: return $T_\ell$
```

Once we have found the $\mu \cdot k$-coreset of points from the CoverTree, we apply an algorithm to retrieve the solution and we compute the objective function. In particular we will use Gonzalez Algorithm 1 and Solve k-Center Algorithm 5 for k-Center Clustering and the strategy adopted in the Section 5.2 for Robust k-Center Clustering.

For all the algorithms used we will include the time to compute the coreset inside the measured computing time to retrieve a solution.

6.6.1 Comparison between k-Center Clustering Algorithms

The first experiment conducted in the field of Clustering is to compare the two algorithms that solve the K-Center Clustering problem, as explained in the previous Section 5.1. Both algorithms will have the same coreset as input to allow for a direct comparison of the two methods in the same environment. This is particularly important when μ is small, as it can significantly affect the results. Clearly, when $\mu = 1$, Algorithm 7 retrieves a coreset with $\geq k$ nodes, making the size of the coreset similar to a feasible k-Clustering. For this reason, the first set of data collected can have the same objective function value as it happens in the METRO dataset, and the two algorithms will differ only in terms of the time required to compute the solution.

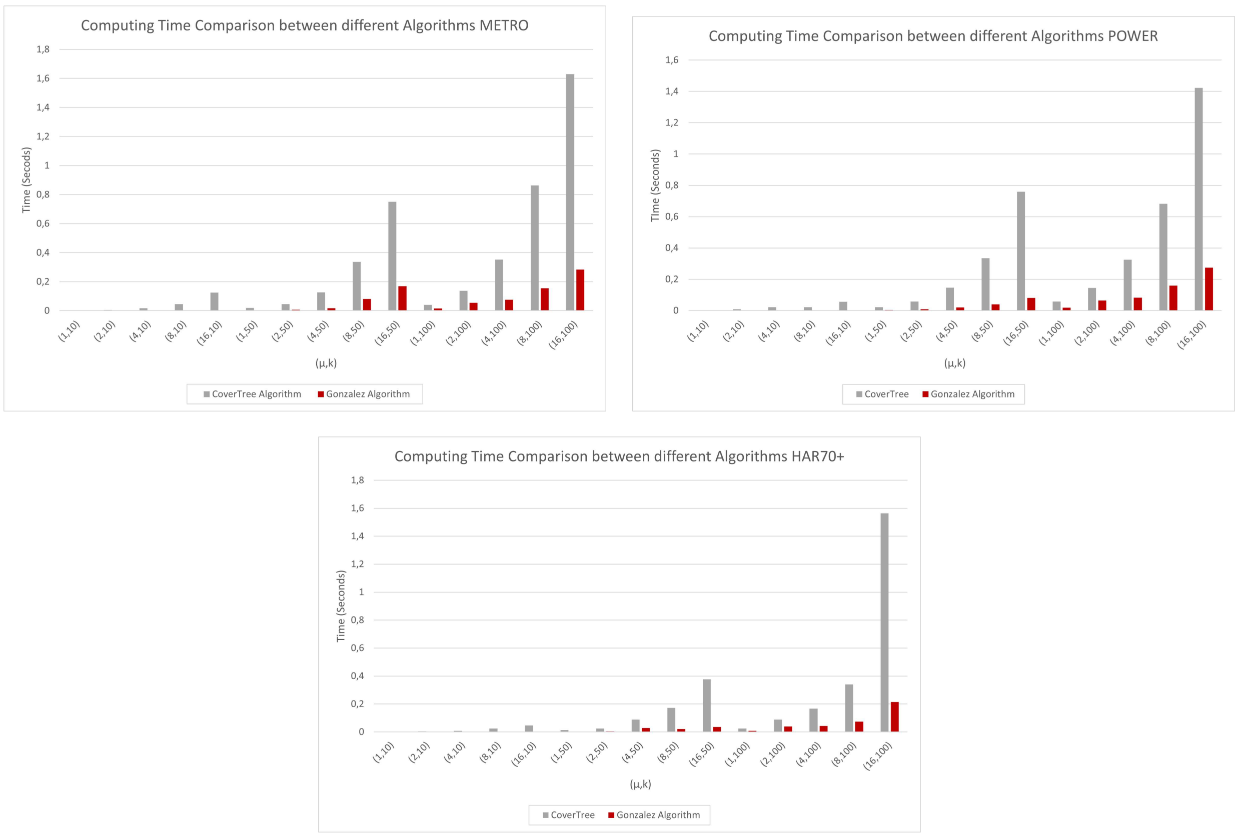


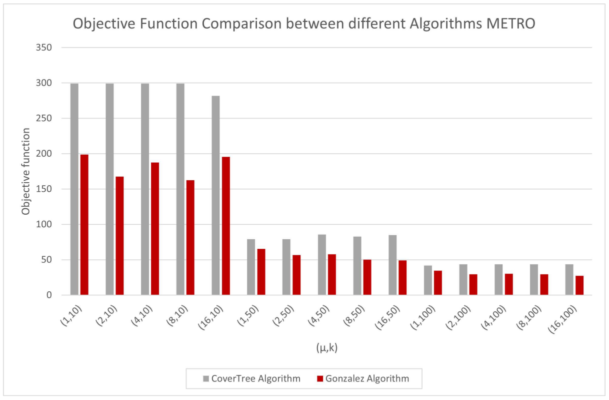
Figure 6.5: A comparison of execution times for the two algorithms that solve the k-Center Clustering problem with $(\mu \cdot k)$-coresets as input, using different pairs (μ, k). The three histograms represent the three input datasets.

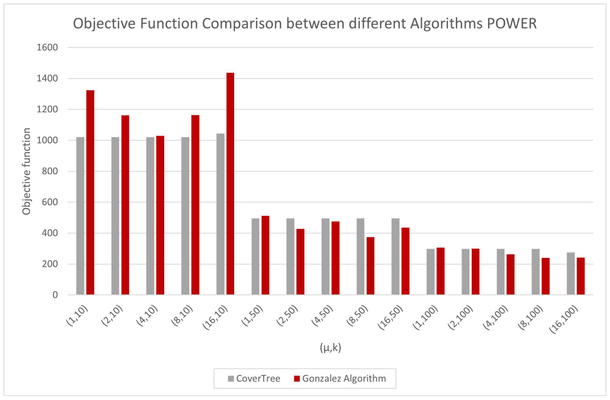
Even though Gonzalez's algorithm has a quadratic dependence on k, contrary to Algorithm 5, which has a linear dependence, it turns out to be faster, since it has been proven that the insertion algorithm in the Cover Tree requires a lot of time.

	Metro	Power	HAR70+
$k = 10$	96%	95%	95%
$k = 50$	82%	86%	81%
$k = 100$	73%	71%	71%

Table 6.6: Percentage of time saved by employing the Gonzalez algorithm instead of Algorithm 5 across various datasets and k.

The percentages in Table 6.6 are calculated as follows: Let $Alg1$ represent Algorithm 5, and $Alg2$ represent the Gonzalez algorithm. The percentage gained is computed using the formula $percentage\_gained = \frac{time_{Alg1} - time_{Alg2}}{time_{Alg1}}$. The table illustrates that the time saved by using the Gonzalez algorithm instead of Algorithm 5 is significant. However, this time-saving tends to diminish as k increases, highlighting the differing dependence of the two algorithms on k.





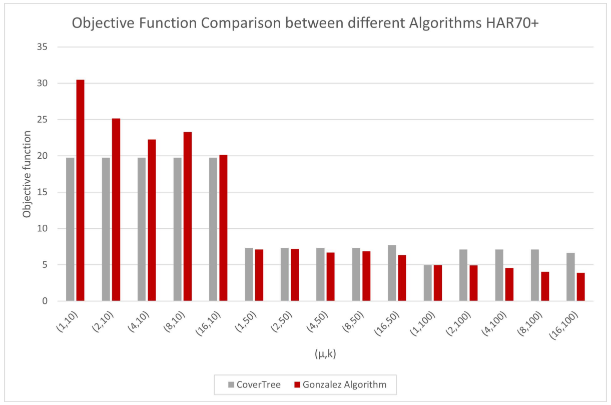


Figure 6.6: A comparison of objective function values for the two algorithms that solve the k-Center Clustering problem with $(\mu \cdot k)$-coresets as input, using different pairs $(\mu,\ k)$. The three histograms represent the three input datasets.

For what concerns the quality of the solutions, Gonzalez's algorithm outperforms the alternative on average. However, contrary to expectations, there are instances where increasing the size of the input coreset, finds in worse solution. This discrepancy could be attributed to either an inadequately representative coreset or an unfavorable selection of the initial center point.

With regard to the quality of the solutions, Gonzalez's algorithm appears to be better on average. However, sometimes, enlarging the coreset could change the solution, resulting in a worse one. This could be the result of a poorly representative coreset.

It is important to highlight that even though, for some combinations of k and μ, the Gonzalez Algorithm performs worse than Algorithm 5 (only in terms of the objective function), especially for the datasets Power and HAR70+ with $k = 10$, for almost every k and for every dataset, the Gonzalez Algorithm finds a better solution than Algorithm 5.

In particular, Solve k-Center Algorithm 5 often returns the same solution, changing it only when the coreset changes significantly. We use Solve k-Center Algorithm 5 with $m = 10$ to produce 10 CoverTrees. Perhaps with a larger m, the algorithm would retrieve different (and hopefully better) solutions, but it would also become significantly slower and would not be comparable to the Gonzalez one in terms of time. The Gonzalez algorithm proves to be more flexible, changing the solution even when the coreset doesn't change much. For example, when we move from a coreset of $n \geq 10$ points (which can be the solution) to a new coreset of $n \geq 20$ nodes with $k = 10$ and $\mu = 2$. The diversity of solutions for Gonzalez is not always a good thing since it may find a worse solution. However, Figure 6.6 shows that using a larger coreset, the algorithm tends to perform better.

6.6.2 Gonzalez Algorithm Testing

Despite Gonzalez having a quadratic dependence on k, we have observed that it performs well in practice. Therefore, another experiment conducted involves comparing the performance of Gonzalez in terms of time and objective function by providing it with two types of input: the coreset retrieved from the tree (with increasing size as μ increases) and the entire dataset. To better illustrate the difference between the coreset approach and using the whole dataset with Gonzalez, we report only the worst and the best solutions when using a coreset as input in terms of optimality.

In the table below, we present the computation times for both the worst and best solutions achieved using the coreset approach, as well as the time required to compute the solution using all the points in the dataset. Please note that the values reported correspond to the time required to compute solutions with both the worst and the best objective functions, rather than representing the smallest and largest times associated with different coreset sizes.

METRO $\sim$ 1.52M Points	Worst Time (Seconds)	Best Time (Seconds)	All points Time (Seconds)
$k = 10$	0.0001	0.0013	14.133
$k = 50$	0.0033	0.1696	80.325
$k = 100$	0.0150	0.2829	165.35

Power $\sim$ 2M Points	Worst Time (Seconds)	Best Time (Seconds)	All points Time (Seconds)
$k = 10$	0.0019	0.0009	22.241
$k = 50$	0.0043	0.0393	120.65
$k = 100$	0.0189	0.1587	393.80

HAR70+ $\sim$ 2.26M Points	Worst Time (Seconds)	Best Time (Seconds)	All points Time (Seconds)
$k = 10$	0.0001	0.0016	21.048
$k = 50$	0.0043	0.0360	112.60
$k = 100$	0.0090	0.2144	240.34

Table 6.7: Comparison of Gonzalez Algorithm Execution Times on Different Datasets. The three input scenarios are: All Points, Best Solution Coreset, and Worst Solution Coreset.

As can be seen, for all three datasets, the computation times for obtaining a solution using the entire set of points or a coreset (with a maximum size used for $\mu = 16$ and $k = 100$) are not comparable. Consequently, the results are presented in tabular form. The time spent by the algorithm to retrieve a solution is very similar when comparing the results across the three datasets, particularly when using all the points as input the time similarity is proportionate.

The graph in Figure 6.7 illustrates the quadratic dependence between k and the computation time required to retrieve a solution using the Gonzalez algorithm. It aligns closely with a polynomial regression model (with $degree = 2$).

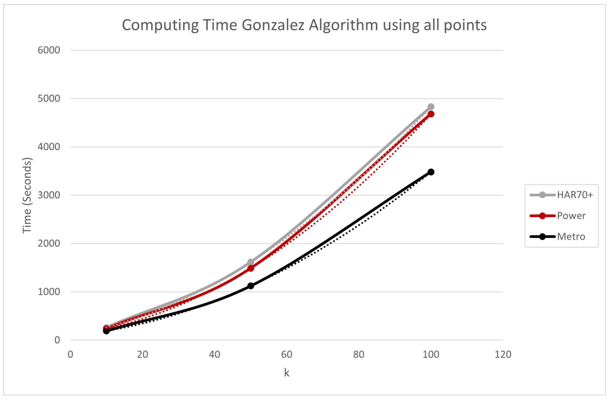


Figure 6.7: Computing time for the Gonzalez algorithm when we use all points as input.

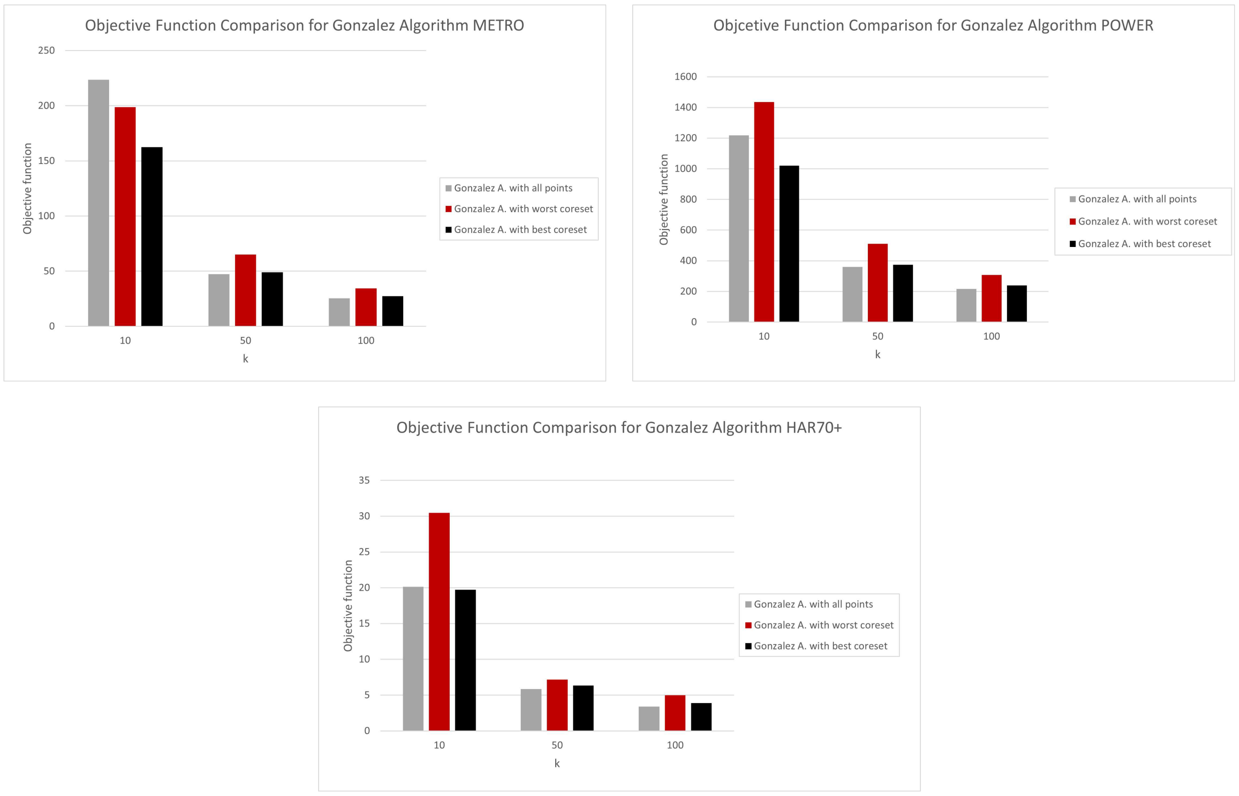


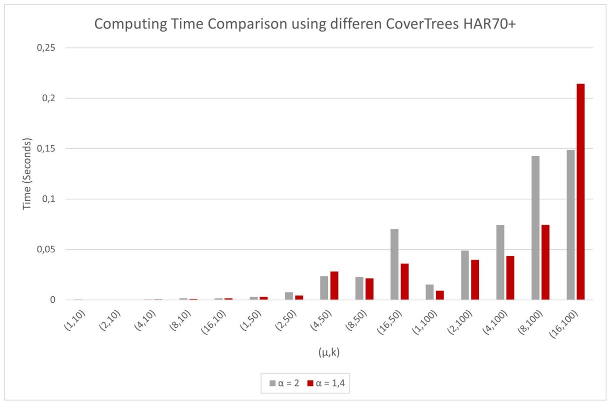
Figure 6.8: Comparison of Gonzalez Algorithm objective function values on Different Datasets. The three input scenarios are: All Points, Best Solution Coreset, and Worst Solution Coreset.

Regarding the quality of the solutions, the results depicted in Figure 6.8 are comparable. For $k = 10$, the Gonzalez Algorithm produces superior solutions for all the datasets when using a coreset, and it does so with significantly less computation time.

However, for $k = \{50, 100\}$, the objective function yields lower values when using the entire set of points as input. It's important to note that, as shown in the previous graphs in Figure 6.6, particularly for Metro and HAR70+, the quality of the solutions improves with a higher μ. Therefore, it's reasonable to expect that a slight increase in μ could lead to even more reliable results while keeping the computation time low.

For the Metro dataset, when $k = 10$, even the first coreset (with $\mu = 10$) retrieved from the Cover Tree produces a better solution than the one found by the Gonzalez Algorithm using all the points as input. In fact, even the worst coreset solution outperforms the All points solution. This highlights the effectiveness of the coreset retrieved from the Tree and the tendency of the Gonzalez Algorithm to perform less optimally when k is smaller. However, this phenomenon does not occur with Power and HAR70+, primarily because the Cover Tree insertion algorithm, and consequently the structure of the Tree, is sensitive to the order in which the points are provided as input.

We also tried to capture the differences between two different Cover Trees derived from the same dataset, more specifically, we use HAR70+ to create two Cover Trees, one with the best possible $\alpha = 1.4$ and another unbalanced Cover Tree with $\alpha = 2$.



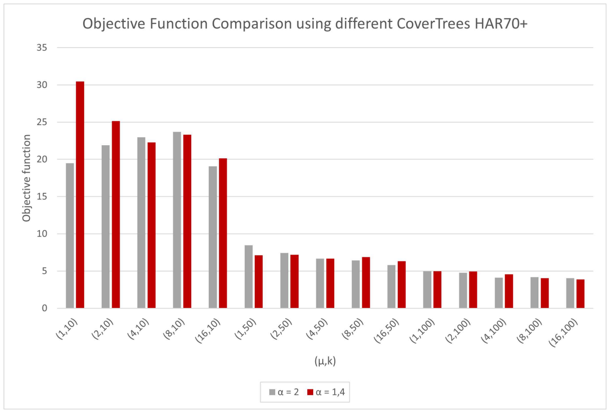


Figure 6.9: A comparison of objective function and time computation values for the Gonzalez algorithm using as input two Cover Trees with $\alpha = \{1.4, 2\}$.

The time computation and the objective function values of the two scenarios are pretty similar. If we consider the average of the values, the Gonzalez Algorithm performs slightly better when the coreset is retrieve from the Cover Tree that has $\alpha = 1.4$.

Due to the significant time constraints associated with data processing (building the tree and calculating the objective function for each clustering), we conducted the experiment using only these two trees.

6.6.3 Robust k- Center Clustering Testing

As with the k-Center problem with z outliers, in line with our previous approach, we input a coreset of increasing size. However, because the method for retrieving the solution, as explained in Section 5.2, relies on ε and $\hat{\varepsilon}$, we had to make adaptations for our experiments.

We opted not to fix the value of ε because $r = \alpha^{\ell_{\max}}/(1+\varepsilon)^i$ also depends on α, and the ideal α varies for each dataset. Keeping ε fixed would result in a trade-off: increasing it would degrade solution quality due to the increasing difference between r_i and r_{i+1}, while decreasing it would extend the time to reach the first non-feasible solution. Please note that as r decreases, W increases.

To tackle this challenge, we implemented a binary search approach. We began with $r_0 = \alpha^{\ell_{\max}}$ and calculated $r_i = r_{i-1}/(1+\varepsilon)$ with varying ε. Initially, we set $\varepsilon = \alpha - 1$, yielding $r = \alpha^{\ell_{\max}}/(\alpha)^i$. Upon reaching the first unfeasible solution (where $W = \sum_{s \in S'} w_s > z$), we adjusted ε to $(-\alpha+1)/(1+\alpha)$, which increases r_i by multiplying it by $(1+\alpha)/2$. This reduces the difference between r_i and r_{i+1}, with $r_{i+1} > r_i$, given that $(1+\alpha)/2 > 0$. By increasing r, after a certain number of multiplications, we will find a new feasible solution. Once a new feasible solution is identified, r_i decreases again by setting $\varepsilon = (\alpha-1)/4$, minimizing the difference between r_i and r_{i+1}. We select the feasible solution with the smallest i as the final outcome.

To sum up:

$$r_i = \left\{ \begin{matrix} \alpha^{\ell_{\max}} & i = 0 \\ r_{i-1}/(1+\varepsilon) & otherwise \end{matrix} \right\}$$

$$\varepsilon = \{\alpha - 1, (-\alpha+1)/(1+\alpha), (\alpha-1)/4\}$$

To set $\hat{\varepsilon}$, we followed the approach outlined in paper [11]. This involved using a slightly different algorithm, where the only variance is in the conditions at lines 5 and 8. Specifically, we employed the ModifiedOutliersCluster algorithm.

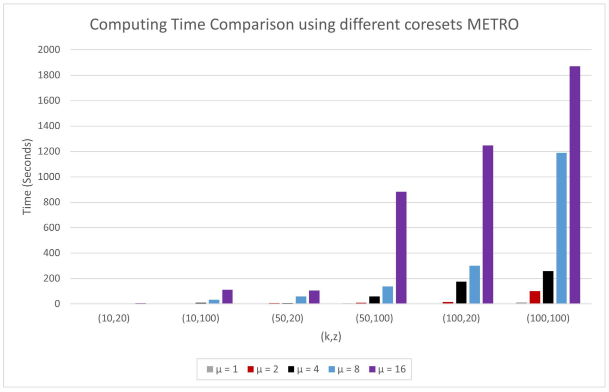
Algorithm 8 ModifiedOutliersCluster

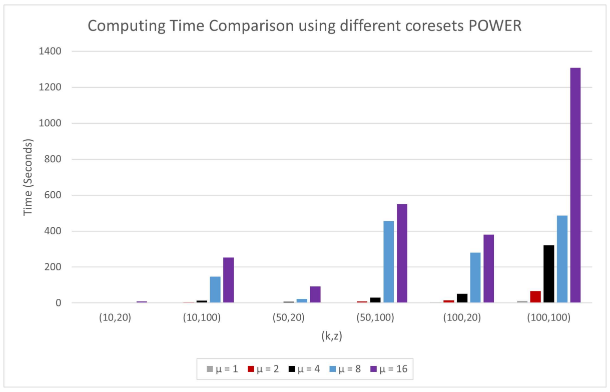
```
Input: A set of points $S$, an integer $k > 0$, $r, \hat{r} > 0$
Output: A set of Centers $C$ of size $k$ and a set of outliers $S'$
1: $S' \leftarrow S$
2: $C \leftarrow \emptyset$
3: while $|C| < k$ AND $S' \neq \emptyset$ do
4:     for $s \in S$ do
5:         $B_s \leftarrow \{v : v \in S' \wedge d(v,s) \leq (r + 2\hat{r})\}$
6:     $c \leftarrow \arg\max_{s \in S} \sum_{v \in B_s} w_v$
7:     $C \leftarrow C \cup \{c\}$
8:     $E_c \leftarrow \{v : v \in S' \wedge d(v,c) \leq (3r + 4\hat{r})\}$
9:     $S' \leftarrow S' \setminus E_c$
10: return $C, S'$
```

The last parameter to set is then $\hat{r}$, that was set to 0 since it gives the best performances in terms of time and objective function. In practice we take into account only the variability of r, that it equal to set $\hat{\varepsilon} = 0$ in OutliersCluster Algorithm or to use the Charikar's Algorithm [2].

The code remains flexible to give the possibility to choice a different $\hat{r}$ so to maintain an approach independent from approximation factors ε and $\hat{\varepsilon}$ changing only $\hat{r}$.

The following graphs illustrate the differences in using the approach described in Section 5.2 with various coreset sizes as input, in terms of computation time and the objective function.





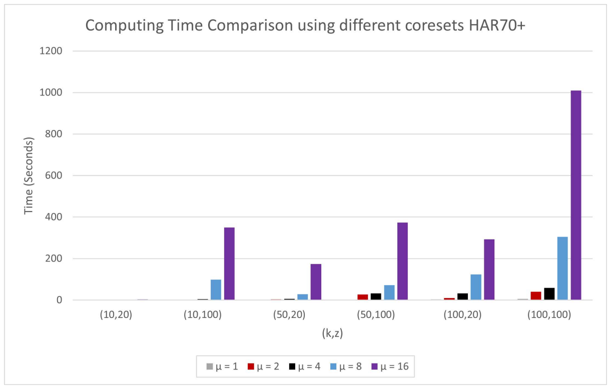
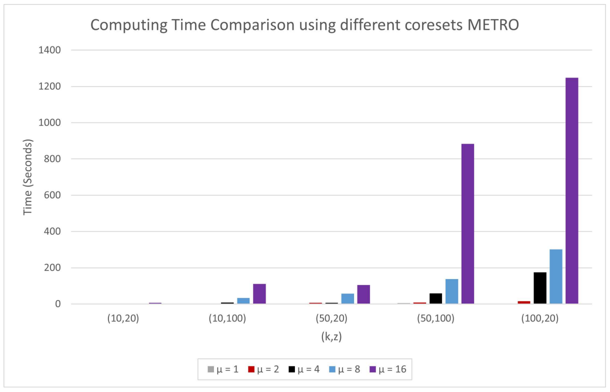


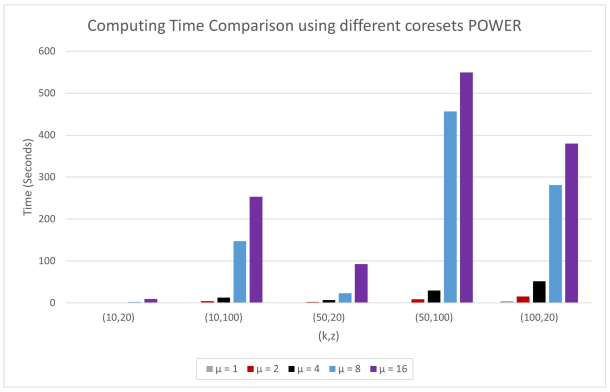
Figure 6.10: A comparison of time computing values for the computation of the coreset and the ModifiedOutliersCluster algorithm using different μ. The experiment was repeated for the three datasets.

In all three datasets, the time required by our approach (comprising the computation of the coreset and multiple executions of the ModifiedOutliersCluster algorithm) increases as the coreset size grows. When the input set is the largest coreset, denoted as Q, with a size of $|Q| \geq 16 \cdot (100 + 100) = 3200$ points, the computation time of the method becomes significantly longer and is not comparable to other scenarios. Even with a relatively small subset of points compared to the entire dataset, the method becomes highly time-intensive.

Figure 6.11 presents the same results as Figure 6.10, excluding the case ($k = 100$, $z = 100$), to provide a clearer view of the results with other combinations of k and z.

The results not only demonstrate that increasing the coreset size leads to longer computation times but also reveal that the OutliersCluster algorithm's performance is sensitive to k. For instance, there is the pair like (50, 100) for METRO that produce larger coreset sizes than others (100, 20) for METRO, yet the solution is computed in less time.





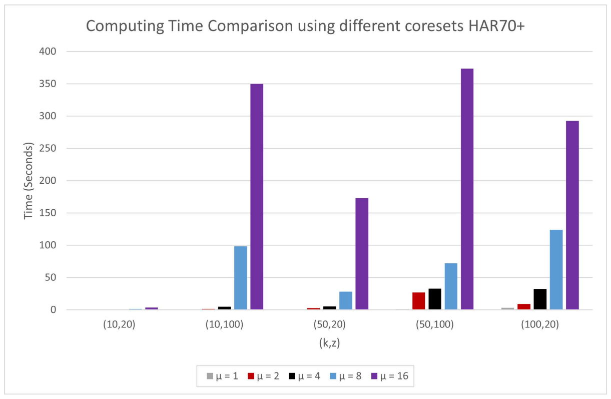


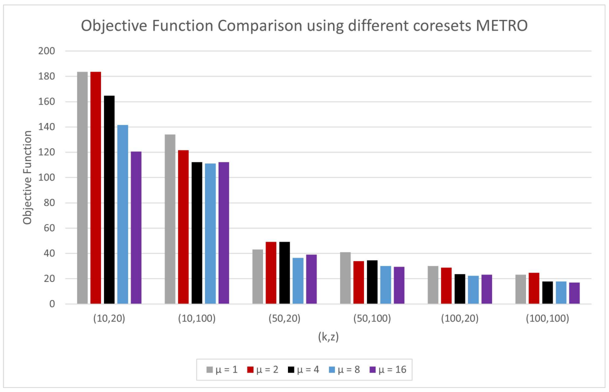
Figure 6.11: A comparison of time computing values for the computation of the coreset and the MODIFIED-OUTLIERSCLUSTER algorithm using different μ without the pair ($k = 100$, $z = 100$). The experiment was repeated for the three datasets.

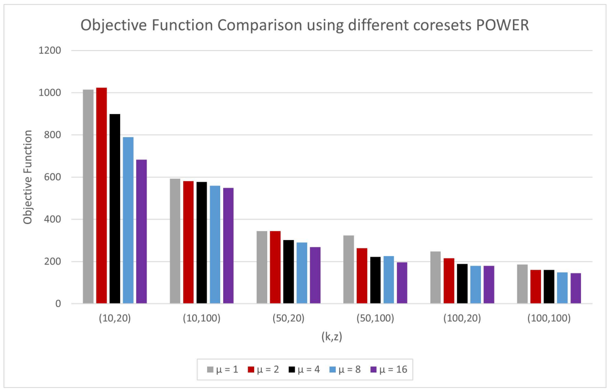
This sensitivity is reasonable because, as explained in [11], the OUTLIERSCLUSTER algorithm retrieves a solution in $O(|Q|^2 + k \cdot |Q|)$ time, where Q is the input set. The algorithm's time complexity is not directly dependent on z, even though $|Q|$ depends on both k and z. In cases where $k_1 + z_1 > k_2 + z_2$ and $k_2 > k_1$, it's possible for $|Q_{k_1,z_1}|$ to be equal to $|Q_{k_2,z_2}|$. This is due to the tree structure of the CoverTree. In such situations, $|Q_{k_2,z_2}|^2 + k_2 \cdot |Q_{k_2,z_2}| > Q_{k_1,z_1}|^2 + k_1 \cdot |Q_{k_1,z_1}|$.

Despite such cases, it's important to note that the method used is primarily dependent on the size of the input set, as demonstrated in the HAR70+ and POWER results.

The results depicted in Figure 6.12 align with expectations. In most cases, the quality of the solution improves linearly with the coreset size, with a few exceptions possibly related to how well the coreset represents the set of points.

Computation time is highly sensitive to the input set's size, but for pairs with larger k and z, the difference in objective function is not substantial. Therefore, the results recommend using a larger coreset when dealing with small k and z to enhance solution quality while maintaining reasonable computation times. For pairs with $k \geq 100$, it's typically sufficient to use $\mu = 4$, which provides a good solution (similar to $\mu = 16$) while requiring a more manageable amount of time.





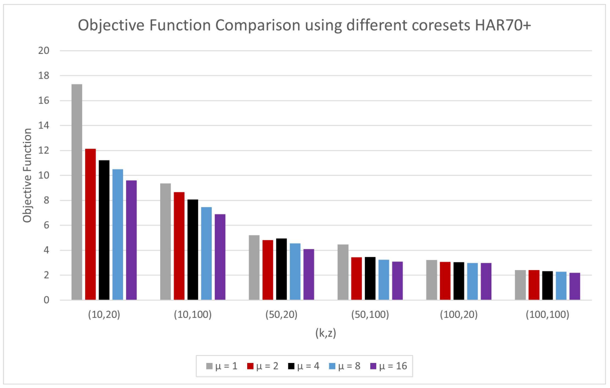


Figure 6.12: A comparison of objective function values for the computation of the coreset and the ModifiedOutliersCluster algorithm using different μ. The experiment was repeated for the three datasets.

Chapter 7

Conclusions

The k-Center Clustering problem is a pivotal and extensively studied challenge in computer science and optimization. Its relevance extends to numerous real-world scenarios, particularly in machine learning, where the volume of information and data plays a pivotal role.

In practical settings, input data often undergo changes, making the task of consistently achieving high-quality solutions challenging. A fully static approach necessitates re-running the algorithm every time the dataset is updated, while a fully dynamic approach can become prohibitively time-consuming for large inputs.

To address this issue, we have adopted a hybrid approach that involves using a representative subset of the input data, known as a coreset. We employ this coreset as input for a static clustering method. When the coreset effectively captures the essential characteristics of the data and remains small in size, the static clustering algorithm can yield high-quality results without excessive time demands. Consequently, the repetition of the clustering method, even across multiple iterations, remains time-efficient.

Our work also explores the concept of finding an optimal coreset by leveraging the Cover Tree data structure, initially introduced by Beygelzimer et al. in 2006 [5]. While earlier proofs were found to be incorrect [9], we have derived new performance bounds for the insertion and deletion algorithms, adapting insights from Ceccarello et al. [11] to the Generalized (α, β)-Cover Tree.

The objective of our research is to assess the quality of clustering solutions when using a coreset of increasing size retrieved from the Cover Tree in comparison to solutions based on the entire set of input points.

We began by inserting all the points into the Cover Tree. However, when using the Standard Cover Tree (with $\alpha = 2, \beta = 1$), insertion times became unfeasible. To address this issue, we tuned the α parameter since, according to the generalized Cover Tree definition, it has a more significant impact than β. We observed that each dataset performed better with a different α, and if the input dataset required a smaller α, the insertion and deletion algorithms became slower. Since a smaller α leads to a decrease in the degree of nodes, in line with the time complexity of the two algorithms, we can infer that a smaller α is preferable for more complex datasets with a higher doubling dimension D.

Although the developed Cover Tree data structure performs poorly when it comes to inserting or deleting many points (our dataset contains approximately 2 million points), we conducted a detailed analysis of the algorithm and identified a weakness related to the computation of all cover sets. Notably, there is a logarithmic dependence between the size of the Cover Tree and the time required to insert a point, which is promising.

It's important to emphasize that our primary objective is not to create a high-performance data structure, but rather to investigate whether, when all the points are contained within the Cover Tree, the solution to the clustering problem using a small coreset retrieved from the tree is comparable to a static algorithm's solution when applied to the entire dataset. Therefore, the fact that computation times do not significantly increase with the size of the Cover Tree is crucial. Even for large Cover Trees, the cost of inserting or deleting points remains relatively consistent.

Thanks to the hierarchical tree structure, we can extract a coreset by selecting a specific level of the tree. Our initial experiment involved comparing two static algorithms for solving the k-Center problem: Solve k-Center Algorithm 5 and Gonzalez Algorithm 1. We measured the overall time required to retrieve the coreset and compute the solution for both methods, and Gonzalez Algorithm consistently proved to be faster. Despite its quadratic time dependence on k, compared to the linear time dependence of Solve k-Center Algorithm 5, Gonzalez Algorithm also consistently produced better solutions in terms of the objective function.

Once we determined that Gonzalez's algorithm is the best choice, we proceeded to compare its performance when the input consists of a relatively small set of points versus using all the points in the dataset. As expected, when employing the coreset approach, the computation time is significantly shorter compared to using all the points. However, the crucial result is that the quality of the solutions is comparable. This underscores the challenge of Gonzalez's algorithm when k is small, as the selection of the initial center becomes more critical, and with a smaller set of points, it exhibits less variability.

The choice of an appropriate α value, which affects the resulting Cover Tree, also has a slight influence on the results. The best α consistently produces better results for both tree update operations and clustering algorithms. It's worth noting that, as Gonzalez [8] demonstrated, achieving an approximation of $2 - \varepsilon$ is impossible in general metrics. Therefore, if the time required to compute a solution significantly decreases when using a coreset while maintaining a comparably good solution to one obtained with all the points, it's a significant achievement.

We followed a similar approach for the k-Center with z outliers problem. Specifically, we extracted a coreset of increasing size from the Cover Tree and then found a solution using the method described in Section 6.6.3. The results demonstrate that computation time increases exponentially as the coreset size grows, while even with a small coreset, we can obtain a good approximate solution. Given the complexity of the problem and the method used, computing the solution using all the points would be infeasible. This further supports our thesis that the use of the Cover Tree to extract a coreset is a viable strategy in terms of both time and solution quality, even for complex algorithms.

While algorithms utilizing coversets from the Cover Tree as input perform well, the time required for initial setup in the Cover Tree remains quite high, even with the relatively small datasets used in our experiments. For the k-Center problem, this approach becomes advantageous primarily when the Gonzalez algorithm is executed multiple times. In such cases, after the initial Cover Tree is created, our approach is convenient when there are subsequent updates that involve only a few points.

It's worth noting that the time computations were measured on a single device, and the computation time for the Gonzalez Algorithm is relatively high as well. It would be interesting to repeat the same experiments on multiple machines to mitigate congestion and reduce variance in time computations. Furthermore, our implementation of the Cover Tree data structure and all the algorithms primarily relies on standard C++ libraries. The primary focus of our work was on theory rather than achieving the best possible implementation performance.

The same approach used for the k-Center problem, both with and without outliers, can also be applied to other clustering problems, such as the matroid center problem and the diversity maximization problem presented in [6]. The strength of the coreset extracted from the Cover Tree lies in its versatility, as it can be effectively employed for a wide range of problems and complex solvers. Even with a small size, these coresets provide a reliable representation of the dataset.

Bibliography

[1] S. Shukri, H. Faris, I. Aljarah, S. Mirjalili, and A. Abraham, “Evolutionary static and dynamic clustering algorithms based on multi-verse optimizer,” *Engineering Applications of Artificial Intelligence*, vol. 72, pp. 54–66, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197618300629

[2] M. Charikar, S. Khuller, D. Mount, and G. Narasimhan, “Algorithms for facility location problems with outliers,” in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001, pp. 642–651.

[3] P. Berkhin, “A survey of clustering data mining techniques,” Grouping Multidimensional Data, 2006.

[4] D. Mount, “Greedy approximation algorithms: The k-center problem,” September 2017.

[5] A. Beygelzimer, S. M. Kakade, and J. Langford, “Cover trees for nearest neighbor,” in *Proceedings of the 23rd International Conference on Machine Learning (ICML ’06)*, 2006.

[6] P. Pellizzoni, A. Pietracaprina, and G. Pucci, “Fully dynamic clustering and diversity maximization in doubling metrics,” *Proceedings of the 18th International Symposium on Algorithms and Data Strructures, WADS 2023, Montreal, Canada, pp. 620-636*, 2023.

[7] A. Gupta, R. Krauthgamer, and J. R. Lee, “Bounded geometries, fractals, and low-distortion embeddings,” in *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2003, pp. 534–543.

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

[9] Y. Elkin, “A new compressed cover tree for k-nearest neighbor search and the stable-under-noise mergegram of a point cloud,” Ph.D. dissertation, University of Liverpool, 2022.

[10] Y. Gu, Z. Napier, Y. Sun, and L. Wang, “Parallel cover trees and their applications,” in *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 259–272. [Online]. Available: https://doi.org/10.1145/3490148.3538581

[11] P. A. P. G. Ceccarello, M., “Solving k-center clustering (with outliers) in mapreduce and streaming, almost as accurately as sequentially,” *Proceedings of the VLDB Endowment*, vol. 12, no. 7, pp. 766–778, 2019.

[12] A. Logacjov and A. Ustad, “HAR70+,” UCI Machine Learning Repository, 2023, DOI: https://doi.org/10.24432/C5CW3D.

[13] G. Hebrail and A. Berard, “Individual household electric power consumption,” UCI Machine Learning Repository, 2012, DOI: https://doi.org/10.24432/C58K54.

[14] V. B. R. R. Davari, Narjes and J. Gama, “MetroPT-3 Dataset,” UCI Machine Learning Repository, 2023, DOI: https://doi.org/10.24432/C5VW3R.

[15] R. A. Fisher, “Iris,” UCI Machine Learning Repository, 1988, DOI: https://doi.org/10.24432/C56C76.