A $(1 + \epsilon)$-Approximation for Ultrametric Embedding in Subquadratic Time

Gabriel Bathie[*]   Guillaume Lagarde[†]

March 18, 2025

Abstract

Efficiently computing accurate representations of high-dimensional data is essential for data analysis and unsupervised learning. Dendrograms, also known as ultrametrics, are widely used representations that preserve hierarchical relationships within the data. However, popular methods for computing them, such as *linkage* algorithms, suffer from quadratic time and space complexity, making them impractical for large datasets. The "best ultrametric embedding" (a.k.a. "best ultrametric fit") problem, which aims to find the ultrametric that best preserves the distances between points in the original data, is known to require at least quadratic time for an exact solution. Recent work has focused on improving scalability by approximating optimal solutions in subquadratic time, resulting in a $(\sqrt{2}+\epsilon)$-approximation (Cohen-Addad, de Joannis de Verclos and Lagarde, 2021).

In this paper, we present the first subquadratic algorithm that achieves arbitrarily precise approximations of the optimal ultrametric embedding. Specifically, we provide an algorithm that, for any $c \geq 1$, outputs a c-approximation of the best ultrametric in time $\tilde{O}(n^{1+1/c})$. In particular, for any fixed $\epsilon > 0$, the algorithm computes a $(1 + \epsilon)$-approximation in time $\tilde{O}(n^{2-\epsilon+o(\epsilon^2)})$.

Experimental results show that our algorithm improves upon previous methods in terms of approximation quality while maintaining comparable running times.

1 Introduction

Clustering is a fundamental technique in data analysis that is used to group similar data points. It helps in uncovering underlying patterns, segmenting data into meaningful categories, and simplifying data representation. Applications of clustering span various domains, including for example bioinformatics, market segmentation, social network analysis, image processing, feature learning, spatial and geoscience, and many others; we refer to [22] for a comprehensive list of references.

While 'flat' clustering methods, such as k-means, can effectively partition data into distinct groups, they often fall short when dealing with complex data

---

[*]LaBRI, University of Bordeaux, France. `https://perso.ens-lyon.fr/gabriel.bathie/`
[†]LaBRI, University of Bordeaux, France. `https://guillaume-lagarde.github.io/`

structures, in particular data points that exhibit multiscale structures. These methods typically assume a fixed number of clusters or rely on specific distance thresholds, which may not capture the true nature of the data. Consequently, they can struggle to reveal the intricate relationships between data points.

Hierarchical clustering addresses these limitations by building a multi-level hierarchy of clusters. This approach does not require specifying the number of clusters a priori and can reveal the nested structure of the data at all levels of granularity. Hierarchical clustering iteratively splits or merges data points based on similarity, forming a tree-like representation known as a dendrogram. This tree structure provides a comprehensive view of the data's organization, allowing for more nuanced interpretations and insights.

Dendrograms and hierarchical clusterings are formalized and quantified using the mathematical concept of *ultrametric*[1]. Ultrametrics provides a rigorous foundation for hierarchical clustering and allows the development and analysis of efficient algorithms to compute and analyze the hierarchical structure of data, see for example [16] or [5].

The most popular methods for constructing an ultrametric are the *agglomerative algorithms*, such as single linkage, complete linkage, average linkage, and Ward's method. These algorithms work *bottom-up* and build an ultrametric by iteratively merging the closest clusters based on a distance metric. While widely used and successful in many applications, they suffer from two major limitations: first, it is not clear what objective functions these methods aim to optimize, making them difficult to analyze and lacking approximation guarantees; second, they have at least quadratic time and space complexity, making them impractical for handling large datasets.

In this paper, we consider the problem of computing the best ultrametric fit (BUF_∞), which is quantified by how well an ultrametric preserves the distances of the original metric using the notion of *maximum distortion*. Formally, given a set X of points in Euclidean space, our goal is to find an ultrametric Δ such that for all $x, y \in X$, $\|x - y\|_2 \leq \Delta(x, y) \leq c \cdot \|x - y\|_2$, where c is as small as possible. This problem was first introduced by Farach et al. [13], who proved that it cannot be solved in subquadratic time and provided an exact algorithm matching this lower bound.

To overcome this quadratic time barrier, Cohen-Addad et al. [10] and Cohen-Addad et al. [11] initiated the development of faster algorithms for computing *approximations* of the optimal ultrametric. They proposed respectively algorithms achieving a $5c$-approximation and a $\sqrt{2}c$-approximation in time $\tilde{O}(n^{1+12/c^2})$, thus providing subquadratic algorithms to approximate BUF_∞ up to factors of ≈ 17.32 and ≈ 4.90, respectively.

1.1 Our contribution

The fundamental question explored in this paper is whether **we can achieve arbitrarily precise approximations of the optimal ultrametric fit in subquadratic time**, or if there exists a theoretical barrier preventing this.

We answer this question positively by constructing the first subquadratic algorithm that achieves arbitrarily precise approximations of the optimal solution

---

[1]An ultrametric is a metric that satisfies a stronger triangle inequality: for any three points, the distance between any two points is at most the maximum of the distances between the other two pairs.

to BUF_∞. More precisely, we prove the following theorem:

Theorem 1. *For any $\gamma \geq 1$ and $\alpha > 1$, there exists an algorithm that computes a $\gamma \cdot \alpha$-approximation of BUF_∞ in time $\tilde{O}(n^{1+1/\gamma^2} + n^{1+1/\alpha^2})$ and space $\tilde{O}(n^{1+1/\gamma^2} + n^{1+1/\alpha^2})$.*

Previously, the best known subquadratic time algorithm for BUF_∞, by Cohen-Addad et al. [11], could only handle approximation factors greater than $\sqrt{2} \cdot \sqrt{12} \approx 4.90$, while we can now achieve arbitrarily precise approximations. Our algorithm is based on two main components:

- An algorithm[2] for computing a γ-Kruskal Tree (abbreviated γ-KT) in time $\tilde{\mathcal{O}}(n^{1+1/\gamma^2})$. This algorithm might be of independent interest. Unlike previous algorithms for γ-KT, it does not require the construction of a γ-spanner, which cannot be built in subquadratic time for $\gamma < \sqrt{2}$ [4].

- A new data structure that computes an α-approximation of the so-called cut weights in time $\tilde{\mathcal{O}}(n^{1+1/\alpha^2})$ (See Definition 3 for the definition of the cut weights). The previous best subquadratic time algorithm was only able to handle approximation factors greater than $\sqrt{2}$, see [11]. Our data structure is based on a dynamic version of the approximate farthest neighbor data structure developed by Pagh et al. [23].

1.2 Experimental results

To complement our theoretical results and to demonstrate the practical efficiency of our algorithm, we perform an extensive set of experiments. We measure the performance of our algorithm both in terms of approximation factor and running time on five classical real-world datasets, and evaluate its scalability on large synthetic datasets. We compare our algorithm with the state-of-the-art algorithm of Cohen-Addad et al. [11] and the widely used implementation of the `fastcluster` Python package. The results show that our algorithm yields better approximations than the algorithm of Cohen-Addad et al. [11] while maintaining a comparable running time and that it can scale to datasets containing millions of points.

1.3 Related work

Carlsson and Mémoli [5] established important foundations for hierarchical clustering, in particular through an in-depth study of ultrametrics that revealed the theoretical properties of hierarchical clustering algorithms.

Other works have explored the complexity of optimizing other distortion measures, such as the average distortion (ℓ_1 norm) and more generally, the ℓ_p norm for different values of p. The problem is NP-complete for $p = 1, 2$ and APX-hard for $p = 1$ (see [25] and [1]). Ailon and Charikar [2] investigated the case of the ℓ_p norm for various values of p and provided polynomial-time algorithms to $O((\log n \log \log n)^{1/p})$-approximate both the best ultrametric and tree metric embeddings using an LP formulation and rounding techniques. These studies consider problems that are NP-hard (at least for $p = 1, 2$) and provide

---

[2]An algorithm was proposed in [10] and [11], but it operates in subquadratic time only when $\gamma \geq \sqrt{12}$

approximation algorithms, but do not focus on scalability: their algorithms have a complexity of $\Omega(n^4)$.

Subquadratic time algorithms in high-dimensional settings, such as those in our work, have been studied by [14] and [7], who provide near-linear time algorithms in the best running case. However, these algorithms lack approximation guarantees for their outputs.

Another line of research focuses on different objective functions to quantify the quality of hierarchical clustering. For instance, Dasgupta [12] introduced an objective function based on cluster properties. Since then, numerous efforts have been devoted to developing algorithms that optimize or approximate this and related metrics, see e.g. [24], [6] and [8].

Significant work has also been done to understand the guarantees provided by popular algorithms. Recently, [9] and [21] proved that average linkage has a small constant approximation ratio for the dual of Dasgupta's objective function, while other methods, such as the bisecting k-means top-down approach, perform poorly for the same objective.

1.4 Organization of the Paper

First, we introduce the necessary definitions and notations in Section 2. In Section 3 we present the high-level algorithm of [10] and [11], on which our work is based, and highlight our improvements. Next, we describe in detail our algorithms to compute a γ-KT and an α-approximation of the cut weights for any $\gamma \geq 1, \alpha \geq 1$ in Section 4 and Section 5 respectively. Finally, we discuss our experimental results in detail in Section 6.

2 Preliminaries

We use the standard complexity definition of $\tilde{O}(\cdot)$, which hides polylogarithmic factors in the input size. Formally, a function $f(n)$ is in $\tilde{O}(g(n))$ if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n) \cdot \log^k n$ for some $k \in \mathbb{N}$ and all large enough $n \in \mathbb{N}$.

For any dimension $d \in \mathbb{N}$, we denote by $\ell_2 : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$ the Euclidean distance between two points $x, y \in \mathbb{R}^d$. Given a set X, an ultrametric distance Δ on X is a distance function $\Delta : X \times X \to \mathbb{R}^+$ where the triangle inequality is replaced by the stronger ultrametric inequality:

$$\forall x, y, z \in X, \Delta(x, z) \leq \max(\Delta(x, y), \Delta(y, z)).$$

In this case, (X, Δ) is an ultrametric space. When X is a finite set, an ultrametric can be represented by a tree T together with a weight function $w : T \to \mathbb{R}^+$, such that

- each element of X is assigned to a leaf of T,

- the weight of each leaf is 0,

- the weights are decreasing along any path from the root to a leaf.

The ultrametric distance induced by T and w is denoted by Δ_T and is defined as $\Delta_T(u, v) = w(\text{LCA}(u, v))$, where $\text{LCA}(u, v)$ represents the least common

ancestor of u and v in T. Using a tree representation of an ultrametric is useful for visualization and interpretation. Specifically, the tree structure can be viewed as a hierarchical clustering of data points, with each node in the tree representing a cluster formed by the leaves of the subtree rooted at that node. A cut through the tree corresponds to a partitioning of the data points into clusters, with cuts made at different levels of the tree yielding clusterings of different granularities. See Fig. 1 for an illustration. This representation has significant utility and numerous applications; see for example the extensive survey of Murtagh and Contreras [22].
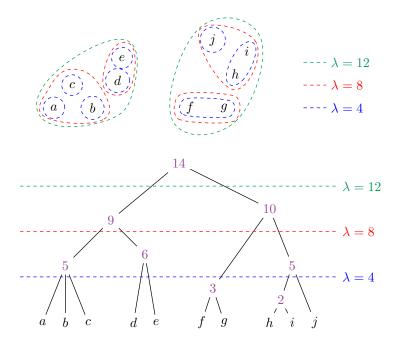

Figure 1: Points and clusters at three different levels of granularity (top) and the corresponding dendrogram (bottom).

Distortion ratio. To quantify how well an ultrametric Δ preserves the distances of a metric ℓ, we use the standard concept of *distortion*. Given a metric space (X, ℓ) and an ultrametric Δ on X such that $\ell(x, y) \leq \Delta(x, y)$ for all $(x, y) \in X^2$, the distortion of Δ, denoted by DIST_∞,[3] is the maximum ratio between the Euclidean distance and the ultrametric distance, i.e.,

$$\mathrm{DIST}_\infty = \max_{(x,y)\in X^2, x\neq y} \frac{\Delta(x,y)}{\ell(x,y)}.$$

---

[3]Similarly, one can define, for any $p \in [1, \infty[$, DIST_p as the p-norm of the vector $\left(\frac{\Delta(u,v)}{\ell(u,v)}\right)_{(u,v)\in X^2}$. The parameter p has a significant influence on the complexity of the problem. For example, when $p = 2$, the best ultrametric fit problem becomes NP-hard, while for $p = \infty$ the problem can be solved in polynomial time.

Best ultrametric fit. As defined by Farach et al. [13], the best ultrametric fit problem (BUF_∞) consists in finding, given a metric space (X, ℓ), an ultrametric Δ on X that preserves the distances as well as possible. Formally, the problem is to find an ultrametric Δ such that:

- $\ell(x, y) \leq \Delta(x, y)$ for all $(x, y) \in X^2$,

- the distortion ratio $\text{DIST}_\infty(\ell, \Delta)$ is minimized.

We denote by DIST_∞^* the (unique) distortion ratio of an optimal solution.

Δ is a *c-approximation* of the best ultrametric fit if the distortion of Δ is at most c times the distortion of the optimal ultrametric, or equivalently if for any pair of points $x, y \in X$, we have:

$$\ell(x, y) \leq \Delta(x, y) \leq c \cdot \text{DIST}_\infty^* \cdot \ell(x, y).$$

The scalar c is called the approximation factor of Δ. By extension, an algorithm is a *c-approximation algorithm* of BUF_∞ if it outputs an ultrametric which is a *c*-approximation of the optimal ultrametric embedding.

Working in Euclidean spaces. From now on, and as in [10] and [11], we consider the case where X is a set of n points in a Euclidean space $\mathbb{R}^d$ equipped with the Euclidean metric ℓ_2, which is one of the most natural settings for many applications in data analysis and unsupervised learning. Note that by the Johnson-Lindenstrauss lemma [17], the dimension d can always be reduced to $O(\log n)$ while preserving the distances between pairs of points up to a multiplicative factor of $(1 + \epsilon)$ for any fixed $\epsilon > 0$.

3 High level algorithm from [10]

We build our approximation algorithm using the framework of Cohen-Addad et al. [10], who provide a way to compute a $5 \cdot \gamma$-approximation of BUF_∞ in time $O(nd + n^{1+12/\gamma^2})$. This result was later improved by Cohen-Addad et al. [11] who provide, for any fixed $\epsilon > 0$, a $(\sqrt{2} + \epsilon) \cdot \gamma$-approximation for the same asymptotic running time. We briefly recall the main ideas of their approach and pinpoint where we improve upon it. We first need a few more definitions.

Definition 2. *Let $G = (V, E, w)$ be a weighted graph and let $\gamma \geq 1$. A spanning tree $T = (V, E_T)$ of G is a γ-Kruskal tree (or γ-KT for short) of G if for every edge $e \in E \setminus E_T$, we have*

$$w(e) \geq \frac{1}{\gamma} \max_{e' \in P_T(e)} w(e'),$$

where $P_T(e)$ is the unique path in T from one endpoint of e to the other.

Let T be a spanning tree of the complete graph induced by the set of points X. For an edge $e = (x, y) \in T$, we denote by $L(e)$ (respectively $R(e)$) the connected component of $T \setminus \{e' \in T \mid \ell_2(e') \leq \ell_2(e)\}$ that contains x (respectively y).

Definition 3. *The cut weight $\text{CW}(e)$ of an edge e is defined as the maximal distance between a point in $L(e)$ and a point in $R(e)$:*

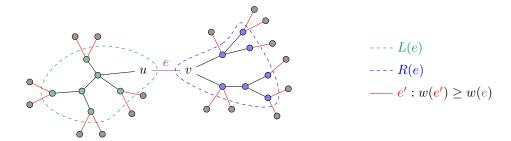$$\text{CW}(e) = \max_{x \in L(e), y \in R(e)} \ell_2(x, y).$$

Figure 2: Illustration of the connected components defined by an edge e. The cut weights is the maximal distance between a point in $L(e)$ and a point in $R(e)$.

The function $\mathrm{CW} : e \mapsto \mathrm{CW}(e)$ for $e \in E_T$ is referred to as the *cut weights of T*. See Fig. 2.

We say that ACW is an α-approximation of the cut weights of T if for every edge e of T, we have

$$\mathrm{CW}(e) \leq \mathrm{ACW}(e) \leq \alpha \cdot \mathrm{CW}(e).$$

Given a spanning tree T and a edge-weight function $w : E_T \to \mathbb{R}^+$, the *Cartesian tree* of T with respect to w is a weighted binary tree defined inductively as follows:

- If T is a single node, then the Cartesian tree is a single node with weight 0.

- Otherwise, the root of the Cartesian tree corresponds to the edge e of T with the largest weight, and the left and right children are the Cartesian trees of the two connected components of $T \setminus \{e\}$.

Cohen-Addad et al. [10] provide an high-level algorithm that achieves a $\gamma \cdot \alpha$-approximation of BUF_∞ by computing a γ-KT and its α-approximate cut weights. This algorithm is inspired by the one presented by Farach et al. [13], who provide a quadratic time algorithm for computing the best ultrametric embedding of the data. The main difference is that in [10], the first two steps are approximated rather than computed exactly. We outline this algorithm in Algorithm 1.

Algorithm 1: $\gamma \cdot \alpha$-approx. of the best ultrametric fit

---

Input: $X \subseteq \mathbb{R}^d$: set of points
Output: A $\gamma \cdot \alpha$-approximation of BUF_∞
1   $T \leftarrow \gamma$-KT T of the complete graph induced by X
2   $\mathrm{ACW} \leftarrow \alpha$-approximation of the cut weights of T
3   $C_T \leftarrow$ Cartesian tree of T w.r.t. ACW
4   **return** *the ultrametric induced by C_T*

Theorem 4 ([10, Theorem 3.1]). *For any $\gamma \geq 1$ and $\alpha \geq 1$, the output of Algorithm 1 is a $\gamma \cdot \alpha$-approximation of the best ultrametric fit.*

Furthermore step 3 of Algorithm 1 can be easily computed in $O(n \log n)$ time using a disjoint-set data structure [13]. Hence, if there are algorithms that compute a γ-KT in time $f(n, d)$, and an α-approximation of the cut weights in time $g(n, d)$, then there is an algorithm that computes a $\gamma \cdot \alpha$-approximation of the best ultrametric fit in time $O(n \log n) + f(n, d) + g(n, d)$. In this work, we show that there are algorithms with $f(n, d) = \tilde{O}(nd + n^{1+1/\gamma^2})$ and $g(n, d) = \tilde{O}(nd + n^{1+1/\alpha^2})$. As mentioned in the preliminary section, we can assume w.l.o.g. $d = O(\log n)$, hence this term is asymptotically dominated by the other in both cases, and we drop it from now on.

Complexity of computing a γ-KT. Cohen-Addad et al. [10] present an algorithm for computing a γ-KT in time $\tilde{O}\left(n^{1+O(1/\gamma^2)}\right)$. This is achieved by constructing a sparse γ-spanner using the LSH-based algorithm of Har-Peled et al. [15] and then computing a minimum spanning tree of the γ-spanner. However, as explained by [4], the $O(1/\gamma^2)$ term in the exponent contains a factor of 12, so this algorithm is only faster than the exact quadratic algorithm for $\gamma > \sqrt{12}$. Furthermore, Andoni and Zhang [4] prove that γ-spanners cannot[4] be constructed in subquadratic time for $\gamma < \sqrt{2}$, thus making spanner-based approaches inefficient for building γ-KTs when $\gamma < \sqrt{2}$. We address this limitation by introducing a new algorithm that computes a γ-KT in time $\tilde{O}\left(n^{1+1/\gamma^2}\right)$ without relying on spanners.

Complexity of approximating the cut weights.

- Cohen-Addad et al. [10] provide an algorithm that computes a 5-approximation of the cut weights in time $O(nd + n \log n)$.

- Cohen-Addad et al. [11] improve the previous result by using a data structure called coresets to provide an algorithm that computes a $(\sqrt{2} + \epsilon)$-approximation of the cut weights in time

$$
O\left(n \cdot d \cdot \left(\frac{\log \frac{1}{\varepsilon}}{\varepsilon^2} + \frac{\log n}{\varepsilon}\right) + n \cdot \frac{1}{\varepsilon^{4.5}} \log \frac{1}{\varepsilon}\right).
$$

In both cases, the total asymptotic running time to compute the ultrametric is $\tilde{O}(nd + n^{1+12/\gamma^2})$, dominated by the time complexity of the algorithm that computes a γ-KT (or more precisely, a γ-spanner in these algorithms). This observation indicates that there is room to improve the approximation factor of the cut weights without increasing the total running time of the algorithm. However, the methods presented in [10] and [11] cannot leverage this observation due to their inherent (geometric) bottlenecks, with approximation factors of 5 and $\sqrt{2}$ for computing the cut weights, respectively.

Our Contributions The main contributions of this paper are as follows: first, we restore the claim of Cohen-Addad et al. [10] by providing an algorithm that, for any $\gamma \geq 1$, computes a γ-KT in time $\tilde{O}(n^{1+1/\gamma^2} \log \delta)$ (Theorem 5), where δ is the *spread* of the input space X, defined as the ratio between the diameter

---

[4]Unless extra nodes, called non-metric Steiner points, are added to the graph.

and the minimum pairwise distance of X. Second, we present an algorithm that computes an α-approximation of the cut weights in time $\tilde{O}(n^{1+1/\alpha^2})$ for any $\alpha \geq 1$ (Theorem 6).

Theorem 5. *For any $\gamma > 1$, there is an algorithm that computes a γ-KT of a given n-points Euclidean space X in time and space $\tilde{O}(n^{1+1/\gamma^2} \log \delta)$, where δ is the spread of X.*

Theorem 6. *For any $\alpha > 1$, there is an algorithm that computes a α-approximation of the cut weights of an n-nodes tree T in time and space $\tilde{O}(n^{1+1/\alpha^2})$.*

Finally, these two results can be combined to obtain a c-approximation of BUF_∞, as shown in the following corollary.

Corollary 7. *For any $c \geq 1$, there exists an algorithm that computes a c-approximation of BUF_∞ in time $\tilde{O}(n^{1+1/c})$.*

Proof. Take $\gamma = \sqrt{c}$ and $\alpha = \sqrt{c}$ in Theorem 5 and Theorem 6, respectively. By using Theorem 4, we obtain a c-approximation of BUF_∞, and the total running time is $\tilde{O}(n^{1+1/c})$. $\qquad\square$

For $c = (1 + \epsilon)$, this gives a $(1 + \epsilon)$-approximation of BUF_∞ in time $\tilde{O}(n^{2-\epsilon+o(\epsilon^2)})$, which remains subquadratic in n. For comparison, when $c = 1$ (no approximation), it is known that the best ultrametric fit problem cannot be solved in subquadratic time, see for example [13].

High-level overview of the techniques.

- **Theorem 5:** We achieve this result using a new method that avoids γ-spanner constructions. The algorithm operates essentially through a breadth-first traversal of the graph, guided by locality-sensitive hashing of the data.

- **Theorem 6:** This algorithm is built on a dynamic version of the approximate farthest neighbor data structure developed by Pagh et al. [23], which supports queries in time $\tilde{O}(n^{1/\alpha^2})$ and space $\tilde{O}(n^{1+1/\alpha^2})$. We extend this data structure to work over a partition of the input space X, allowing approximate farthest neighbor queries within *clusters* (i.e., subsets of the partition) and efficient merging of clusters.

We now proceed to the details of these two algorithms.

4 Algorithm for γ-Kruskal tree

In this section, we present the algorithm to compute a γ-KT, as stated in Theorem 5, along with its full analysis.

Recall that the algorithm by Cohen-Addad et al. [10] for computing a γ-KT operates in two steps: first, it computes a sparse γ-spanner of the complete graph induced by the set of points X, and then it computes a minimum spanning tree of the γ-spanner. To find a γ-spanner, they use the algorithm of Har-Peled et al. [15], which runs in time $\tilde{\mathcal{O}}(n^{1+12/\gamma^2})$ and uses Locality-Sensitive Hashing. This

algorithm is subquadratic only when $\gamma < \sqrt{12}$. While it might be possible to reduce this constant, Andoni and Zhang [4] showed that spanners cannot be constructed in subquadratic time for $\gamma < \sqrt{2}$.

Instead of relying on spanners, we propose an algorithm that builds a γ-KT for any $\gamma \geq 1$ in time $\tilde{O}(n^{1+1/\gamma^2} \log \delta)$, essentially via a breadth-first traversal guided by LSH.

We first recall the definition of Locality-Sensitive Hash functions (LSH):

Definition 8. *Let (X, ℓ) be a metric space over n points, and let $c > 1$ and $R > 0$. A family $\mathcal{H}$ of functions is a (ρ, c, R)-LSH if, when choosing a function h uniformly at random from $\mathcal{H}$ we have, for every $x, y \in X$:*

- $\ell(x, y) \leq R \Rightarrow \mathbb{P}\left[h(x) = h(y)\right] \geq 1/n^{\rho}$

- $\ell(x, y) \geq cR \Rightarrow \mathbb{P}\left[h(x) = h(y)\right] \leq 1/n$

Andoni and Indyk [3] showed that when X is a Euclidean space, i.e. ℓ is the ℓ_2 metric, then for every $c > 1$ and every $R > 0$, there exists a $(1/c^2, c, R)$-LSH family of functions $\mathcal{H}_{c,R}$, and the evaluation of a random function from $\mathcal{H}_{c,R}$ on all n points of X takes $\tilde{O}(n)$ time.

Given an LSH function h, the *buckets* of h are the equivalence classes of the relation $x \sim y \Leftrightarrow h(x) = h(y)$. Intuitively, points with the same hash value are put into the same bucket.

Finally, we introduce an important property, denoted (*), that will be useful in explaining the behavior of our algorithm.

Definition 9. *We say that a set E of edges satisfies the property (*) if for any pair $(u, v) \in X^2$, there is a path from u to v in E using only edges of weight at most $\gamma \cdot \ell_2(u, v)$.*

Overview of the algorithm.   The algorithm works in two steps:

- First we compute a set E of $\tilde{O}(n^{1+1/\gamma^2} \cdot \log \delta)$ edges that satisfies (*). This is the purpose of Proposition 10.

- Then, we compute a minimum spanning tree of E. Lemma 11 shows that this tree is a γ-KT. This part is done in time $O(|E| \log |E|) = \tilde{O}(n^{1+1/\gamma^2})$ using the standard algorithm of Kruskal [20].

The above algorithm, combined with the following proposition and lemma, yields the main result of this section, a proof of Theorem 5.

Proposition 10. *There is an algorithm that computes a set E of $\tilde{O}(n^{1+1/\gamma^2} \cdot \log \delta)$ edges that satisfies (*) in time $\tilde{O}(n^{1+1/\gamma^2} \log \delta)$.*

Lemma 11. *Let X be an n-points Euclidean space and let $\gamma > 1$. Let $E \subseteq X^2$ be a set of edges that satisfies (*). then running Kruskal's algorithm on E' yields a γ-KT of X in time $O(|E| \log |E|)$.*

Algorithm 2: Local-BFS in a graph

Input: $X \subseteq \mathbb{R}^d$: set of points,
γ: approximation parameter,
R: target radius

1 $E \leftarrow \emptyset$;
2 $h \leftarrow (1/\gamma^2, \gamma, R)$-LSH$(X)$;
3 foreach *bucket B of h* **do**
4   coll $\leftarrow 0$;
5 $S \leftarrow B$;
6   **while** *S is not empty* **do**
7 $x \leftarrow S.\text{pop\_any}()$;
8 $q \leftarrow \text{QUEUE}()$; $q.push(x)$;
9   **while** *q is not empty* **do**
10 $u \leftarrow q.pop()$;
11   **foreach** *v in S* **do**
12   **if** $\ell_2(u, v) \leq \gamma \cdot R$ **then**
13 $S.\text{remove}(v)$;
14 $E \leftarrow E \cup \{(u, v)\}$;
15 $q.push(v)$;
16   **else**
17   coll $\leftarrow$ coll $+ 1$;
18 return E

4.1 Proof of Proposition 10

To prove Proposition 10, we introduce Algorithm 2 that will be used as a useful subroutine in the proof.

We start by proving properties of Algorithm 2. We first show that it runs in expected quasilinear time.

Lemma 12. *Algorithm 2 runs in $\tilde{O}(n)$ expected time.*

Proof. As mentioned above, applying the LSH function h to all points of X takes $\tilde{O}(n)$ time.

We show that for each bucket B of h, the body of the outer `foreach` loop runs in time $O(|B| + \text{collisions}(B))$, where $\text{collisions}(B)$ is the value of `coll` at the end of the `foreach` iteration on bucket B.

By implementing the removal from S as a filtering procedure, both branches of the `if` statement take constant time.

Next, the algorithm enters the body of the `foreach` loop on Line 11 at most $|B| + \text{collisions}(B)$ times, as the loop body either removes a vertex for S, which has size $|B|$, or increases the value of `coll`. Since each vertex is pushed (and therefore popped) from the queue at most once, the outer `while` loop runs in time $O(|B| + \text{collisions}(B))$, and the Algorithm 2 runs in time $\tilde{O}(n + \sum_B \text{collisions}(B))$.

It remains to show that the expected value of $\sum_B \text{collisions}(B)$ is $O(n)$. The variable `coll` is incremented whenever we encounter a pair of vertices u, v that fall into the same bucket and $\ell_2(u, v) > \gamma R$. By definition of the LSH function, this event occurs with probability at most $1/n$, hence over all pairs u, v we have

at most n such collisions in total in expectation, and therefore Algorithm 2 runs in time $\tilde{O}(n)$. $\square$

We now show that the set E of edges computed by Algorithm 2 has a desired connectivity property.

Lemma 13. *Algorithm 2 returns a set E of up to n edges of weight at most γR such that for any pair of points $(u,v) \in X^2$ with $\ell_2(u,v) \le R$ and $h(u) = h(v)$, there is a path from u to v in E.*

Proof. Since we only add to E edges of weight at most γR, the condition on edge weights is satisfied. Furthermore, when adding an edge (u,v) to E, we remove v from S and it is never added back, hence E contains at most n edges.

To prove the connectivity property, we show that the body of the outer `while` loop adds to E a set of edges that spans the subset of vertices u in B that are connected to x with edges from of weight at most γR. Let $G_{\gamma R}$ denote the graph with vertex set X and all edges of length at most γR. First, notice that the inner `while` loop of Algorithm 2 performs a breadth-first traversal, i.e. the vertices are enumerated in order of nondecreasing distance d_G to x, where $d_G(u,v)$ defined as the minimum number of edges in a u–v path in $G_{\gamma R}$, and $+\infty$ is there is no such path.

We show by induction on $d_G(x,u)$ that for every $u \in B$ such that $d_G(x,u)$ is finite, then there is a path from x to u in E. The base case is when $d_G(x,u) = 0$, i.e. $x = u$: the property is trivially satisfied. Next, let $u \in B$ be such that $d_G(x,u) = k+1$. By definition, every neighbor v of u is at distance at least k of x, and at least one of them satisfies $d(x,v) = k$. Consider the first such v enumerated by the algorithm: all other neighbors of u will be enumerated after v, therefore, at this point, u has not been removed from S, hence we add the edge (u,v) to E. By induction hypothesis, there is a path from x to v in E, and the edge (u,v) ensures that there is also a path from x to u in E, concluding our induction. $\square$

Next, we use Algorithm 2 to build a set of edges that connects every pair of points at distance at most R with edges of weight at most γR.

Corollary 14. *Let X be a subset of n points in $\mathbb{R}^d$, let $\gamma > 1$, and let R be a target radius. There is an algorithm that runs in expected time $\tilde{O}(n^{1+1/\gamma^2})$ and, with probability at least $1 - 1/n$, returns a set $E \subseteq X^2$ of at most $\tilde{O}(n^{1+1/\gamma^2})$ edges, each of length at most γR such that for any pair $(u,v) \in X^2$ of distance at most R, there is a path from u to v in E.*

Proof. We run Algorithm 2 $L = 3n^{1/\gamma^2} \log n$ times independently with parameters (γ, R), and return the union of all resulting sets E.

Consider a pair u, v of points of X such that $\ell_2(u,v) \le R$. If at some step $i = 1, \ldots, L$ of the above iteration, u and v are in the same bucket of h, then by Lemma 13, there will be a path from u to v in E. As $\ell_2(u,v) \le R$, the probability that u and v fall in the same bucket at a fixed step i is at least $n^{-\rho}$. Therefore, using independence of the runs, the probability that u and v never fall in the same bucket over all L runs is at most

$$(1 - n^{-\rho})^L \le e^{-3 \log n} = 1/n^3.$$

Furthermore, the algorithm fails only when there exists a pair u, v such that $\ell_2(u, v) \leq R$ that never fall in the same bucket. By union bound over all pairs u, v, this event occurs with probability at most $1/n$.

The running time of this procedure follows from Lemma 12. $\qquad\square$

Proof of Proposition 10. By Corollary 14, there is an algorithm which, when used with a fixed value R, ensures that any two points u, v such that $\ell_2(u, v) \leq R$ are connected by a path of edges of length at most $\gamma \cdot R$ with probability at least $1 - 1/n$.

To create the set E satisfying property (*), we run this algorithm for $O(\log n \cdot \log \delta)$ values of R, in order to cover the range of distances between points in X. More precisely let $d_{\min}$ and $d_{\max}$ denote the minimum and maximum distance between distinct points in X, and let $\delta = d_{\max}/d_{\min}$ denote the spread of X. Let $\tau = 1 + 1/\log n$. By using the algorithm of Corollary 14 with parameter $\gamma' = \gamma/\tau$ and $R = d_{\min}, d_{\min}\tau, d_{\min}\tau^2, \ldots, \tau d_{\max}$, we get a set E that connects any pair of points at distance d with edges of weight $\tau \cdot (\gamma/\tau) \cdot d = \gamma \cdot d$. The number of calls to Corollary 14 is, up to a constant

$$\log_\tau(d_{\max}/d_{\min}) = \frac{\log(\delta)}{\log \tau} = O(\log n \cdot \log(\delta)).$$

The running time of the algorithm of Corollary 14 with parameter γ' is $\tilde{O}(n^{1+1/\gamma'^2})$, which is $\tilde{O}(n^{1+1/\gamma^2})$: this concludes the proof. $\qquad\square$

4.2 Proof of Lemma 11

We are now ready to prove Lemma 11.

Proof of Lemma 11. Kruskal's algorithm builds a minimum spanning tree by first sorting the edges in E in order of non-decreasing weights, and then iterating over all edges in order, adding to T each edge that connects two disjoint connected components of T.

Assume that for the sake of contradiction that the output tree T is not a γ-KT. Then there exists an edge (u, v) of weight $w = \ell_2(u, v)$ such that there is an edge e' of weight $\ell_2(e') > \gamma \cdot w$ on the path from u to v in T. Removing e' from T yields two connected components C_1 and C_2, with $u \in C_1$ and $v \in C_2$. By assumption, there is a $u - v$ path consisting of edges in E', each of weight at most $\gamma \cdot w$. As $u \in C_1$ and $v \in C_2$, one edge e^* of this path is not in T and has one endpoint in C_1 and the other in C_2. As $\ell_2(e^*) < \ell_2(e')$, e^* was considered before e' by the algorithm, i.e. at a time where C_1 and C_2 were disjoint. Therefore, the algorithm has added e^* to T, a contradiction as $e^* \notin T$. $\qquad\square$

5 Better Cut-Weights via Approximate Farthest Neighbors

The second step of Algorithm 1 is to compute the cut weights of the spanning tree obtained in the first step. Farach et al. [13] provided a quadratic-time algorithm to compute the exact cut weights. More recently, [10] and [11] proposed a 5- and a $\sqrt{2}$-approximation algorithm that both operate in quasilinear time. However,

their approximations are inherently limited due to their reliance on specific geometric properties of Euclidean spaces.

In this section, we introduce an α-approximation algorithm for the cut weights that works for any $\alpha > 1$. Our result is as follows:

Theorem 6. *For any $\alpha > 1$, there is an algorithm that computes a α-approximation of the cut weights of an n-nodes tree T in time and space $\tilde{O}(n^{1+1/\alpha^2})$.*

The core ingredient of our algorithm is a *dynamic* version of the data structure for approximate farthest neighbor of Pagh et al. [23], which we present in the next subsection. The algorithm behind the Theorem 6 is described in the second part of this section.

5.1 Dynamic Approximate Farthest Neighbor

In order to obtain an α-approximation of the cut weights, we use the data structure of Pagh et al. [23] for *approximate farthest neighbors* (AFN) in Euclidean spaces. Their data structure preprocesses a subset of a metric space, and can then find in that subset an α-approximate farthest neighbor of a given query point in time $\tilde{O}(n^{1/\alpha^2})$.

Definition 15 (AFN)**.** *Let (X, d) be a metric space and let $\alpha > 1$. A data structure $\mathcal{D}$ solves the α-AFN problem over a set $S \subseteq X$ if, given a point $q \in X$, it returns a point r that is an α-approximate farthest neighbor of q in S, i.e. it holds that*

$$d(q, r) \geq \frac{1}{\alpha} \max_{p \in S} d(q, p).$$

To fit our use case, we show that one can extend the data structure of Pagh et al. [23] to be *dynamic*, i.e. given the data structure for two disjoint subsets S, S' of (X, d), we can construct a data structure for $S \sqcup S'$ faster than the time needed to build it from scratch.

Theorem 16. *There is a data structure for α-AFN over a dynamic partition of a metric space (X, d) of n points that supports the following operations:*

- INITIALIZE(X, α)*: create an data structure containing a cluster $S_x = \{x\}$ for each x in X, in time $\tilde{O}(n^{1+1/\alpha^2})$.*

- QUERY$(\mathcal{D}, S, q)$*: given a cluster S in $\mathcal{D}$ and a query point $q \in X$, return an α-approximate farthest neighbor of q in S in time $\tilde{O}(n^{1/\alpha^2})$.*

- MERGE$(\mathcal{D}, S, S')$*: given two clusters S, S' in $\mathcal{D}$, add the cluster $S'' = S \sqcup S'$ to $\mathcal{D}$ in time $\tilde{O}(n^{1/\alpha^2} \cdot \min(|S|, |S'|, n^{1/\alpha^2}))$. This operation consumes the clusters S and S', i.e. they cannot be used in other operations afterward.*

This data structure uses $\tilde{O}(n^{1+1/\alpha^2})$ space. Furthermore, this construction is probabilistic (in the INITIALIZE function), and for every q and S, the QUERY operation fails with probability at most $1/n^3$.

Before proving Theorem 16, we briefly recall how the approximate farthest neighbor data structure of Pagh et al. [23] works. We will then show how to adapt it to our needs.

Let $p_1, p_2, \ldots, p_n$ be the points in S. Intuitively, the data structure of Pagh et al. uses projections on many random lines to identify points in the input set that are "extremal" along some direction, and search the farthest point of a given query point in this subset of extremal points. More precisely, the data structure samples $L = O(n^{1/\alpha^2})$ Gaussian random vectors $a_1, \ldots, a_L$. Then, for each $i \leq L$ and each $j = 1, \ldots, n$, let β_{ij} denote the value of the inner product $\langle a_i, p_j \rangle$. For each i, the structure stores a collection $\mathcal{C}_i$ containing the (up to) $M = \tilde{O}(n^{1/\alpha^2})$ pairs (j, β_{ij}) that have the largest value of β_{ij}. Storing these indices requires $O(n^{1/\alpha^2} \cdot \min(|S|, n^{1/\alpha^2}))$ space.

While this data structure may store up to $\tilde{O}(n^{2/\alpha^2}$ candidate points, Pagh et al. show how to select a subset S' of M points that depends on the query point q so that with constant probability, S' contains an α-approximate farthest neighbor of q. Under the assumption that for every i, there is an efficient way of iterating over the p_j in order of decreasing values of $\langle a_i, p_j \rangle$, their query algorithm runs in time $\tilde{O}(n^{1/\alpha^2})$.

Note that the randomness for the probability of error is taken over the choice of the a_i's at construction not: once these are fixed, the query algorithm is deterministic. By taking $\Theta(\log n)$ independent copies of this data structure and returning the farthest point from q across all queries, we can reduce the probability of error to less than $1/n^3$.

We are now ready to explain how to build the dynamic data structure of Theorem 16.

Proof of Theorem 16. We first select $L' = L \cdot \Theta(\log n) = \tilde{O}(n^{1/\alpha^2})$ random Gaussian vectors $a_1, \ldots, a_{L'}$. The data structure $\mathcal{D}$ will maintain a dynamic partition of X, and for each *cluster* (i.e. each set) S in the partition, store for every i the $\min(|S|, M)$ points that maximize the value of $\beta_{ij} = \langle a_i, p_j \rangle$.

For the INITIALIZE operation, each cluster contains a single point: we only need to compute the values β_{ij} for every i and j, which takes time $\tilde{O}(n^{1+1/\alpha^2})$ as $d = O(\log n)$.

For every i, we store the collection $\mathcal{C}_i$ of pairs $(j, \beta_{ij})_{j=1,\ldots,M}$ in a binary search tree using β_{ij} as key. As we can efficiently iterate over the elements of a binary search tree in order of decreasing keys, the QUERY operation runs in time $\tilde{O}(n^{1/\alpha^2})$ using the algorithm of Pagh et al. Furthermore, as $L' = L \cdot \Theta(\log n)$, the probability of returning a point that is not a c-approximate neighbor of the query point q is at most $1/n^3$.

Finally, using binary search trees allows us to implement the MERGE operation in time $\tilde{O}(n^{1/\alpha^2} \cdot \min(|S|, |S'|, n^{1/\alpha^2}))$. Let S and S' be two clusters, and assume w.l.o.g. that $|S| \leq |S'|$. To merge S and S', we move all elements from the cluster $\mathcal{C}_i$ of S to the corresponding cluster $\mathcal{C}_i'$ of S', and then truncate $\mathcal{C}_i'$ to keep only the M pairs with largest key. As $|\mathcal{C}_i'| \leq n$, inserting an element into $\mathcal{C}_i'$ takes time $O(\log n)$ and moving the elements for a fixed i takes time $O(|\mathcal{C}_i| \cdot \log n)$. As $\mathcal{C}_i$ contains $O(\min(|S|, n^{1/\alpha^2})$ elements and there are $L' = \tilde{O}(n^{1/\alpha^2})$ collections $\mathcal{C}_i$ to move, this takes a total of $\tilde{O}(n^{1/\alpha^2} \cdot \min(|S|, |S'|, n^{1/\alpha^2}))$ time.

Moving elements instead of copying them ensures that the space total usage remains $O(n^{1+1/\alpha^2})$. □

5.2 Approximate Cut-weight Algorithm

We give a proof of Theorem 6, i.e. we give an algorithm that computes an α-approximation of cut weights of a tree in time and space $\tilde{O}(n^{1+1/\alpha^2})$, using the data structure of Theorem 16.

By augmenting the aforementioned data structure with a disjoint-set data structure, we can additionally support the following operations:

- FIND($\mathcal{D}, q$): return the (unique) cluster S in $\mathcal{D}$ that contains q, in time $O(\log^* n)$.

- ENUMERATE($\mathcal{D}, S$): iterate over all elements of a cluster S in $\mathcal{D}$, in total time $O(|S|)$.

The procedure to compute an α-approximation of the cut weights is given in Algorithm 3.

Algorithm 3: α-approximation of the cut weights

Input: $X \subseteq \mathbb{R}^d$: set of points,
T: spanning tree as a list of weighted edges sorted by non-decreasing weight,
$\alpha > 1$: approximation parameter

1   $\mathcal{D} \leftarrow$ INITIALIZE(X);
2   **foreach** *edge $e = (x, y)$ in increasing order of weights* **do**
3      $S_x \leftarrow$ FIND($\mathcal{D}, x$); $S_y \leftarrow$ FIND($\mathcal{D}, y$);
4   **if** $|S_x| > |S_y|$ **then**
5   Swap S_x and S_y;
6   ACW(e) $\leftarrow \alpha \cdot \max\{w(z, \text{QUERY}(\mathcal{D}, S_y, z)) | z \in \text{ENUMERATE}(\mathcal{D}, S_x)\}$;

7   MERGE($\mathcal{D}, S_x, S_y$);
8   **return** ACW

We now turn to proving Theorem 6. We first analyze the complexity of Algorithm 3. The space complexity of $\tilde{O}(n^{1+1/\alpha^2})$ follows from that of the data structure $\mathcal{D}$ of Theorem 16. To obtain the desired time complexity, we crucially rely on the fact that, on Line 6 of Algorithm 3, we iterate over the smallest of S_x and S_y and query the other when computing ACW. This allows us to prove using a counting argument that the algorithm makes $O(n \log n)$ calls to QUERY, showing that the algorithm runs in time $\tilde{O}(n^{1+1/\alpha^2})$.

Lemma 17. *Algorithm 3 runs in time $\tilde{O}(n^{1+1/\alpha^2})$.*

Proof. First, we show that the body of the `foreach` loop on Line 2 of Algorithm 3 runs in time $\tilde{O}(n^{1/\alpha^2} \cdot \min(|S_x|, |S_y|))$. The two FIND operations on Line 3 run in time $O(\log^* n) = \tilde{O}(n^{1/\alpha^2})$. Then, after Line 5, $|S_x|$ is at most $|S_y|$, hence we show that the rest of the loop body runs in time $\tilde{O}(n^{1/\alpha^2} \cdot |S_x|)$. On Line 6, we make a single call to ENUMERATE, plus one distance computation and one call to QUERY for each $z \in S_x$ which takes a total time of $\tilde{O}(n^{1/\alpha^2} \cdot |S_x|)$. Finally, from Theorem 16, the call to MERGE on Line 7 takes time $\tilde{O}(n^{1/\alpha^2} \cdot \min(|S_x|, |S_y|, n^{1/\alpha^2}))$, which is also dominated by $\tilde{O}(n^{1/\alpha^2} \cdot |S_x|)$.

Now, we show that the sum of $\min(|S_x|, |S_y|)$ over the course of the algorithm is at most $O(n \log n)$. At the start of the algorithm, each point $z \in X$ belongs to a cluster of size 1, and, as T is a spanning tree, they will all be in a single cluster of size n when the procedure ends. If during an iteration of the loop, z was in the smaller cluster, the size of its cluster will increase at least twofold. Therefore, this can happen at most $O(\log n)$ times to each point. Hence, over all points in X, the event "being in the smallest cluster" happens at most $O(n \log n)$ times.

Therefore, the running time of the algorithm is $\tilde{O}(n^{1+1/\alpha^2})$. □

We now show that the ACW function is an α-approximation of the cut weights with high probability. Intuitively, putting aside low-probability errors, $\text{QUERY}(\mathcal{D}, S_y, q)$ returns a point whose distance to q is between $d_{\max}/\alpha$ and $d_{\max}$, where $d_{\max}$ is the maximum distance between q and a point of S_y. Therefore, by taking the maximum over all points in S_x and multiplying by α, we obtain a value between $\text{CW}(e)$ and $\alpha \cdot \text{CW}(e)$.

Lemma 18. *With high probability, the function* ACW *returned by Algorithm 3 is an α-approximation of the cut weights* CW *of T. More precisely, with probability at least $1 - 1/n$, we have, for every $e \in T$:*

$$\text{CW}(e) \leq \text{ACW}(e) \leq \alpha \cdot \text{CW}(e).$$

Proof. First, assume that no call to QUERY returns an incorrect result: we show that in this case, ACW is an α-approximation of the cut weights, and will then show that this event occurs with probability at least $1 - 1/n$. One can show by induction that, at each iteration, we have $S_x = L(e)$ and $S_y = R(e)$ where $e = (x, y)$. Therefore, we can rewrite the cut weight of e as

$$\text{CW}(e) = \max_{z \in S_x} \max_{z' \in S_y} w(z, z').$$

Furthermore, the $\text{QUERY}(\mathcal{D}, S_y, z)$ algorithm returns a point r_z such that $\alpha \cdot w(z, r_z) \geq \max_{z' \in S_y} w(z, z')$. Hence, we have:

$$\begin{aligned}
\text{ACW}(e) &= \alpha \cdot \max_{z \in S_x} w(z, r_z) \\
&\geq \max_{z \in S_x} \max_{z' \in S_y} w(z, z') \\
&= \text{CW}(e)
\end{aligned}$$

On the other hand, $w(z, z_r)$ cannot exceed $\text{CW}(e)$, hence we have $\text{CW}(e) \leq \text{ACW}(e) \leq \alpha \text{CW}(e)$: this shows that ACW is an α-approximation of CW.

The algorithm makes $O(n \log n) = o(n^2)$ calls to QUERY (see the second part of the proof of Lemma 17) therefore by union bound, it errs with probability $o(1/n)$, which is less than $1/n$ for large enough n. □

6 Experiments

We evaluate the performance of our algorithm on two types of datasets. First, as in [10] and [11], we use five classic real-world datasets to evaluate both the quality of the approximation and the runtime of the algorithms; see Table 1 for details on these datasets. Second, we use synthetic datasets to evaluate how the

runtime of the algorithms scales with larger datasets. In this case, we cannot measure the quality of the approximation given by the algorithms, since this takes quadratic time, which is unreasonably long for large datasets. We compare our algorithm with the state-of-the-art algorithm of Cohen-Addad et al. [11] and the widely used implementation of the `fastcluster` Python package.

Dataset	size	dim.	optimal dist.
IRIS	150	4	8.07
DIABETES	768	8	5.96
MICE	1080	77	4.92
PENDIGITS	10992	16	13.86
SHUTTLE	58000	9	29.72

Table 1: Description of the datasets used for evaluation. All datasets are publicly available on the UCI ML Repository [19], or Kaggle [18] for the DIABETES dataset.

Algorithm	MICE		PENDIGITS		SHUTTLE	
	apx.	$T\ (s)$	apx.	$T\ (s)$	apx.	$T\ (s)$
FKW	1	$0.12s$	1	$8.16s$	1	$236.72s$
FastUlt ($c = 4$)	1.88	$0.12s$	1.53	$1.80s$	1.41	$21.39s$
FastUlt ($c = 9$)	3.67	$0.07s$	2.26	$0.79s$	1.85	$7.82s$
FastUlt ($c = 16$)	5.58	$0.04s$	2.79	$0.52s$	2.63	$4.69s$
CVL	3.27	$0.14s$	1.85	$0.76s$	2.41	$10.17s$

Table 2: Comparison of FastUlt with the state-of-the-art algorithm by Cohen-Addad et al. [11], denoted "CVL" in the table. The "apx." column reports the approximation factor, i.e. the distortion of the output ultrametric normalized by the distortion of the optimal ultrametric, given by quadratic-time algorithm by Farach et al. [13], denoted "FKW". Each reported distortion or running time value is the average of 30 runs of the algorithm, all standard deviations are less than 10% for appoximation and 2% for runtimes.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`). The experiments were conducted on identical nodes of the Grid'5000 cluster, running Debian GNU/Linux 5.10.0-28-amd64. The hardware configuration includes an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz and 126GB of RAM. For the experiments, our algorithm is implemented in the Rust programming language, version 1.79.0 (`129f3b996`, 2024-06-10). Our code was compiled in `release` mode.

6.1 Main experiments

Experiment 1: Accuracy of the BUF$_\infty$ algorithm. We measure the performance of our BUF$_\infty$ c-approximation algorithm using our γ-KT and α-ACW algorithms with $\alpha = \gamma = \sqrt{c}$, which we call FastUlt. Based on the

results of Experiment A (see Paragraph 6.2), we modify the α-ACW algorithm to multiply distances by $\sqrt{\alpha}$ instead of α: in practice, this is sufficient to overestimate the cut weights and improves the approximation factor. We evaluate FastUlt for different values of c, and for each value we run the algorithm $t = 30$ times on each of the 5 datasets, for a total of 150 runs per value of c.

The key takeaway from this experiment is that our algorithm performs significantly better than the worst-case approximation factor c. For example, if one wants a 2-approximation of the best ultrametric embedding of a dataset, they can run the algorithm with a larger parameter, e.g. $c = 9$ instead of 2. This approach will greatly reduce the running time and space usage from $\tilde{\mathcal{O}}(n^{1.5})$ to $\tilde{\mathcal{O}}(n^{1.11})$ while still providing a high-quality, low-distortion embedding.

We ran FastUlt for $c = 2, 4, 9, 16, 100, 400$: of the 900 resulting runs, FastUlt failed to over-approximate the cut weights in only 20 runs, which is less than 2.3% of the runs. In addition, the algorithm remains very accurate. For example, when using $c = 9$, the procedure returned 2-approximation of the best ultrametric fit in all but 5 of the 150 runs. The results for $c = 2, 4, 9, 16$ are reported in Fig. 3 (values $c = 100, 400$ are omitted here for readability of the figure).
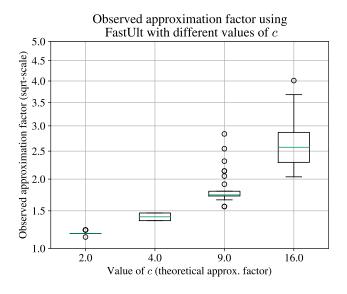

Figure 3: Approximation factor obtained by FastUlt for different values of c.

Experiment 2: Comparison with previous methods. We compare the performance of FastUlt with the previous best known algorithm for BUF$_\infty$ by Cohen-Addad et al. [11], both in terms of the quality of the approximation factor of the best ultrametric and the running time. The results are given in Table 2. These results show that FastUlt with $c = 4$ can achieve better ultrametric embeddings than the algorithm of Cohen-Addad et al. [11] for a comparable computational cost. Furthermore, by using $c = 9$ or 16, we can obtain embeddings with similar distortion but with a lower running time. Thus, another advantage of FastUlt is the ability to trade off approximation factor

and running time by adjusting the parameter c, for any desired embedding quality.

Experiment 3: Scaling. Finally, to evaluate how well our algorithm scales to larger datasets, we create two synthetic datasets with up to one million data points. These datasets contain N uniformly random points from $[0,1]^d$, for $(N, d) \in [(10^5, 100), (10^6, 10)]$. The running times are given in Table 3.

Algorithm	$N = 10^5$ $d = 100$	$N = 10^6$ $d = 10$
FASTULT ($c = 4$)	1m 39.89s	21m 58.50s
FASTULT ($c = 9$)	34.07s	5m 13.99s
FASTULT ($c = 16$)	19.24s	2m 34.97s
CVL	7.96s	1m 54.48s
Single Linkage	9m 55.36s	$\geq$ 10h

Table 3: Running time of the FASTULT algorithm, the algorithm of Cohen-Addad et al. [11] and the *single linkage* algorithm from the `fastcluster` python package on datasets of N d-dimensional random points. Each reported running time is an average of 30 runs.

We observe that, while our algorithm is slower than that of Cohen-Addad et al. [11], FASTULT does not suffer from the same quadratic blow-up as classical algorithms such as single linkage, and maintains a reasonable running time that can be afforded for the analysis of large datasets.

6.2 Additional experiments

Experiment A: Accuracy of the α-approximate cut weights (ACW) algorithm. On Line 6 of the approximate cut weights algorithm (Algorithm 3), we multiply the distance to the approximate farthest neighbor by α to ensure that $\text{ACW}(e)$ is at least $\text{CW}(e)$ for every edge e. However, our AFN data structure turns out to be very accurate in practice: it often returns points whose distance to the query point q is close to maximal. Therefore, the multiplication by α artificially increases the approximation factor of the algorithm[5]: multiplying by another number $c < \alpha$ would suffice in practice.

To quantify this phenomenon, we compute the approximation factor when multiplying by the smallest constant c^* (the same for all queries) such that $\text{ACW}(e)$ is at least $\text{CW}(e)$ for every e, instead of multiplying by α. To only measure the effect of α-ACW algorithm on the approximation factor, we run this algorithm on an exact minimum spanning tree instead of a γ-KT. The optimal distortion is computed using the algorithm of Farach et al. [13]. The results are reported in Fig. 4.

In practice, the observed approximation factor is much lower than the theoretical approximation factor: 75% of the time, the observed approximation factor is less than 1.5, and it is always less than 3, even for $\alpha = 20$. This means that,

---

[5]Recall that the approximation factor of the algorithm is the ratio of the distortion of the computed ultrametric divided by the distortion of the optimal ultrametric.
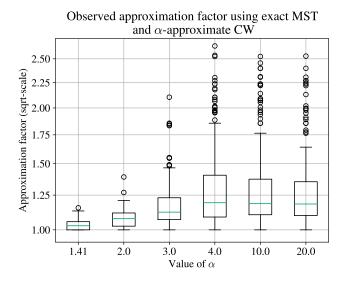
Figure 4: Accuracy of the α-approximate cut weights algorithm. For each value of α, the algorithm is run 30 times on each of the 5 datasets, resulting in 150 data points per boxplot.

for $\alpha = 20$, the α-ACW algorithm could multiply the distances by 3 instead of α and still obtain an over-approximation of the cut weights.

Experiment B: Accuracy of the γ-KT algorithm. Next, we measure the accuracy of the γ-KT algorithm. We run the exact cut weights algorithm on the output of the γ-KT algorithm, and compare the distortion of the resulting ultrametric to the optimal ultrametric (given by the algorithm of Farach et al. [13]). The results are given in Table 4: again we observe that the γ-KT algorithm performs much better than the theoretical guarantee.

MST algorithm	Theoretical approx. factor	Observed approx. factor
1.41-KT	1.41	1.06 ± 0.08
2.0-KT	2.0	1.23 ± 0.23
3.0-KT	3.0	1.63 ± 0.47
4.0-KT	4.0	1.87 ± 0.64
10.0-KT	10.0	3.37 ± 2.21
20.0-KT	20.0	5.22 ± 5.57

Table 4: Accuracy of the γ-KT algorithm with exact cut weights. For each value of γ, the average and standard deviation of the approximation factor are computed over 30 runs of the algorithm on each of the 5 datasets, for a total of 150 data points.

References

[1] Richa Agarwala, Vineet Bafna, Martin Farach, Mike Paterson, and Mikkel Thorup. On the approximability of numerical taxonomy (fitting distances by tree metrics). *SIAM Journal on Computing*, 28(3):1073–1085, 1998.

[2] Nir Ailon and Moses Charikar. Fitting tree metrics: Hierarchical clustering and phylogeny. *SIAM Journal on Computing*, 40(5):1275–1291, 2011.

[3] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 459–468, 2006.

[4] Alexandr Andoni and Hengjie Zhang. Sub-quadratic $(1+\epsilon)$-approximate euclidean spanners, with applications. *arXiv preprint arXiv:2310.05315*, 2023.

[5] Gunnar E. Carlsson and Facundo Mémoli. Characterization, stability and convergence of hierarchical clustering methods. *J. Mach. Learn. Res.*, 11: 1425–1470, 2010. doi: 10.5555/1756006.1859898. URL https://dl.acm.org/doi/10.5555/1756006.1859898.

[6] Moses Charikar and Vaggos Chatziafratis. Approximate hierarchical clustering via sparsest cut and spreading metrics. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–854. SIAM, 2017.

[7] Michael Cochez and Hao Mou. Twister tries: Approximate hierarchical agglomerative clustering for average distance in linear time. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, pages 505–517, 2015.

[8] Vincent Cohen-Addad, Varun Kanade, and Frederik Mallmann-Trenn. Hierarchical clustering beyond the worst-case. *Advances in Neural Information Processing Systems*, 30, 2017.

[9] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-Trenn, and Claire Mathieu. Hierarchical clustering: Objective functions and algorithms. *Journal of the ACM (JACM)*, 66(4):1–42, 2019.

[10] Vincent Cohen-Addad, Karthik C. S., and Guillaume Lagarde. On efficient low distortion ultrametric embedding. *CoRR*, abs/2008.06700, 2020. URL https://arxiv.org/abs/2008.06700.

[11] Vincent Cohen-Addad, Rémi de Joannis de Verclos, and Guillaume Lagarde. Improving ultrametrics embeddings through coresets. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 2060–2068. PMLR, 2021. URL http://proceedings.mlr.press/v139/cohen-addad21a.html.

[12] Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 118–127, 2016.

[13] Martin Farach, Sampath Kannan, and Tandy J. Warnow. A robust model for finding optimal evolutionary trees. *Algorithmica*, 13(1/2):155–179, 1995. doi: 10.1007/BF01188585. URL https://doi.org/10.1007/BF01188585.

[14] Sean Gilpin, Buyue Qian, and Ian Davidson. Efficient hierarchical clustering of large high dimensional datasets. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1371–1380, 2013.

[15] Sariel Har-Peled, Piotr Indyk, and Anastasios Sidiropoulos. Euclidean spanners in high dimensions. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 804–809. SIAM, 2013.

[16] Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[17] William B Johnson, Joram Lindenstrauss, and Gideon Schechtman. Extensions of lipschitz maps into banach spaces. *Israel Journal of Mathematics*, 54(2):129–138, 1986.

[18] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. Diabetes dataset - kaggle. https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database, 2024. Accessed: 2024-06-23.

[19] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. The UCI machine learning repository. http://archive.ics.uci.edu, 2024. URL https://archive.ics.uci.edu. Accessed: 2024-06-23.

[20] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[21] Benjamin Moseley and Joshua R Wang. Approximation bounds for hierarchical clustering: Average linkage, bisecting k-means, and local search. *Journal of Machine Learning Research*, 24(1):1–36, 2023.

[22] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: an overview, II. *WIREs Data Mining Knowl. Discov.*, 7(6), 2017. doi: 10.1002/WIDM.1219. URL https://doi.org/10.1002/widm.1219.

[23] Rasmus Pagh, Francesco Silvestri, Johan Sivertsen, and Matthew Skala. Approximate furthest neighbor with application to annulus query. *Information Systems*, 64:152–162, 2017.

[24] Aurko Roy and Sebastian Pokutta. Hierarchical clustering via spreading metrics. *Journal of Machine Learning Research*, 18(88):1–35, 2017.

[25] HT Wareham. On the complexity of inferring evolutionary trees. *Technical Report Technical Report*, 9301, 1993.