Efficient Construction of Probabilistic Tree Embeddings*†

Guy E. Blelloch[1], Yan Gu[2], and Yihan Sun[3]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
`guyb@cs.cmu.edu`

2 **Carnegie Mellon University, Pittsburgh, PA, USA**
`yan.gu@cs.cmu.edu`

3 **Carnegie Mellon University, Pittsburgh, PA, USA**
`yihans@cs.cmu.edu`

Abstract

In this paper we describe an algorithm that embeds a graph metric (V, d_G) on an undirected weighted graph $G = (V, E)$ into a distribution of tree metrics (T, D_T) such that for every pair $u, v \in V$, $d_G(u, v) \leq d_T(u, v)$ and $\mathbf{E}_T[d_T(u, v)] \leq O(\log n) \cdot d_G(u, v)$. Such embeddings have proved highly useful in designing fast approximation algorithms, as many hard problems on graphs are easy to solve on tree instances. For a graph with n vertices and m edges, our algorithm runs in $O(m \log n)$ time with high probability, which improves the previous upper bound of $O(m \log^3 n)$ shown by Mendel et al. in 2009.

The key component of our algorithm is a new approximate single-source shortest-path algorithm, which implements the priority queue with a new data structure, the *bucket-tree structure*. The algorithm has three properties: it only requires linear time in the number of edges in the input graph; the computed distances have a distance preserving property; and when computing the shortest-paths to the k-nearest vertices from the source, it only requires to visit these vertices and their edge lists. These properties are essential to guarantee the correctness and the stated time bound.

Using this shortest-path algorithm, we show how to generate an intermediate structure, the approximate dominance sequences of the input graph, in $O(m \log n)$ time, and further propose a simple yet efficient algorithm to converted this sequence to a tree embedding in $O(n \log n)$ time, both with high probability. Combining the three subroutines gives the stated time bound of the algorithm.

We also show a new application of probabilistic tree embeddings: they can be used to accelerate the construction of a series of approximate distance oracles.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases Graph Algorithm, Metric Embeddings, Probabilistic Tree Embeddings, Single-source Shortest-paths

Digital Object Identifier 10.4230/LIPIcs.ICALP.2017.26

* The full version of this paper is available at arXiv:1605.04651 [7], `https://arxiv.org/abs/1605.04651`.

† This research was supported in part by NSF grants CCF-1314590 and CCF-1533858, and the Intel Science and Technology Center for Cloud Computing.

1 Introduction

The idea of probabilistic tree embeddings [4] is to embed a finite metric into a distribution of tree metrics with a minimum expected distance distortion. A distribution $\mathcal{D}$ of trees of a metric space (X, d_X) should minimize the expected stretch ψ so that:

1. dominating property: for each tree $T \in \mathcal{D}$, $d_X(x, y) \le d_T(x, y)$ for every $x, y \in X$, and
2. expected stretch bound: $\mathbf{E}_{T\sim\mathcal{D}}[d_T(x, y)] \le \psi \cdot d_X(x, y)$ for every $x, y \in X$,

where $d_T(\cdot, \cdot)$ is the tree metric, and $\mathbf{E}_{T\sim\mathcal{D}}$ draws a tree T from the distribution $\mathcal{D}$. After a sequence of results [2, 3, 4], Fakcharoenphol, Rao and Talwar [17] eventually proposed an elegant and asymptotically optimal algorithm (FRT-embedding) with $\psi = O(\log n)$.

Probabilistic tree embeddings facilitate many applications. They lead to practical algorithms to solve a number of problems with good approximation bounds, for example, the k-median problem, buy-at-bulk network design [8], and network congestion minimization [30]. A number of network algorithms use tree embeddings as key components, and such applications include generalized Steiner forest problem, the minimum routing cost spanning tree problem, and the k-source shortest paths problem [22]. Also, tree embeddings are used in solving symmetric diagonally dominant (SDD) linear systems. Classic solutions use spanning trees as the preconditioner, but recent work by Cohen et al. [14] describes a new approach to use trees with Steiner nodes (e.g. FRT trees).

In this paper we discuss yet another remarkable application of probabilistic tree embeddings: constructing of approximate distance oracles (ADOs)—a data structure with compact storage ($o(n^2)$) which can approximately and efficiently answer pairwise distance queries on a metric space. We show that FRT trees can be used to accelerate the construction of some ADOs [24, 34, 10].

Motivated by these applications, efficient algorithms to construct tree embeddings are essential, and there are several results on the topic in recent years [12, 22, 8, 25, 21, 19, 6]. Some of these algorithms are based on different parallel settings, e.g. share-memory setting [8, 19, 6] or distributed setting [22, 21]. As with this paper, most of these algorithms [12, 22, 25, 21, 6] focus on graph metrics, which most of the applications discussed above are based on. In the sequential setting, i.e. on a RAM model, to the best of our knowledge, the most efficient algorithm to construct optimal FRT-embeddings was proposed by Mendel and Schwob [25]. It constructs FRT-embeddings in $O(m \log^3 n)$ expected time given an undirected positively weighted graph with n vertices and m edges. This algorithm, as well as the original construction in the FRT paper [17], works hierarchically by generating each level of a tree top-down. However, such a method can be expensive in time and/or coding complexity. The reason is that the diameter of the graph can be arbitrarily large and the FRT trees may contain many levels, which requires complicated techniques, such as building sub-trees based on quotient graphs.

Our results. The main contribution of this paper is an efficient construction of the FRT-embeddings. Given an undirected positively weighted graph $G = (V, E)$ with n vertices and m edges, our algorithm builds an optimal tree embedding in $O(m \log n)$ time. In our algorithm, instead of generating partitions by level, we adopt an alternative view of the FRT algorithm in [22, 8], which computes the potential ancestors for each vertex using ***dominance sequences*** of a graph (first proposed in [12], and named as least-element lists in [12, 22]). The original algorithm to compute the dominance sequences requires $O(m \log n + n \log^2 n)$ time [12]. We then discuss a new yet simple algorithm to convert the dominance sequences to an FRT tree only using $O(n \log n)$ time. A similar approach was taken by Khan et al. [22] but their output is an implicit representation (instead of an tree) and under the distributed setting and it is not work-efficient without using the observations and tree representations introduced by Blelloch et al. in [8].[1]

Based on the algorithm to efficiently convert the dominance sequences to FRT trees, the time complexity of FRT-embedding construction is bottlenecked by the construction of the dominance sequences. Our efficient approach contains two subroutines:

- An efficient (approximate) single-source shortest-path algorithm, introduced in Section 3. The algorithm has three properties: linear complexity, distance preservation, and the ordering property (full definitions given in Section 3). All three properties are required for the correctness and efficiency of constructing FRT-embedding. Our algorithm is a variant of Dijkstra's algorithm with the priority queue implemented by a new data structure called *bucket-tree structure*.
- An algorithm to integrate the shortest-path distances into the construction of FRT trees, discussed in Section 4. When the diameter of the graph is $n^{O(1)}$, we show that an FRT tree can be built directly using the approximate distances computed by shortest-path algorithm. The challenge is when the graph diameter is large, and we proposed an algorithm that computes the approximate dominance sequences of a graph by concatenating the distances that only use the edges within a relative range of $n^{O(1)}$. Then we show why the approximate dominance sequences still yield valid FRT trees.

With these new algorithmic subroutines, we show that the time complexity of computing FRT-embedding can be reduced to $O(m \log n)$ w.h.p. for an undirected positively weighted graph with arbitrary edge weight.

In addition to the efficient construction of FRT trees, this paper also discuss a new application. We show that FRT trees are intrinsically Ramsey partitions (definition given in Section 5) with asymptotically tight bound, and can achieve even better (constant) bounds on distance approximation. Previous construction algorithms of optimal Ramsey partitions are based on hierarchical CKR partitions, namely, on each level, the partition is individually generated with an independent random radius and new random priorities. In this paper, we present a new proof to show that the randomness in each level is actually unnecessary, so that only one single random permutation is enough and the ratio of radii in consecutive levels can be fixed as 2. Our FRT-tree construction algorithm therefore can be directly applied to a number of different distance oracles that are based on Ramsey partitions and accelerates the construction of these distance oracles.

2 Preliminaries and Notations

Let $G = (V, E)$ be a weighted graph with edge lengths $l : E \to \mathbb{R}_+$, and $d(u, v)$ denote the shortest-path distance in G between nodes u and v. Throughout this paper, we assume that $\min_{x \neq y} d(x, y) = 1$. Let $\Delta = \frac{\max_{x,y} d(x,y)}{\min_{x \neq y} d(x,y)} = \max_{x,y} d(x, y)$, the diameter of the graph G.

In this paper, we use the single source shortest paths problem (SSSP) as a subroutine for a number of algorithms. Consider a weighted graph with n vertices and m edges, Dijkstra's algorithm [15] solves the SSSP in $O(m + n \log n)$ time if the priority queue of distances is maintained using a Fibonacci heap [18].

A premetric (X, d_X) defines on a set X and provides a function $d : X \times X \to \mathbb{R}$ satisfying $d(x, x) = 0$ and $d(x, y) \geq 0$ for $x, y \in X$. A metric (X, d_X) further requires $d(x, y) = 0$ iff $x = y$, symmetry $d(x, y) = d(y, x)$, triangle inequality $d(x, y) \leq d(x, z) + d(z, y)$ for

[1] A simultaneous work by Friedrichs et al. proposed an $O(n \log^3 n)$ algorithm of this conversion (Lemma 7.2 in [19]).

$x, y, z \in X$. The shortest-path distances on a graph is a metric and is called the graph metric and denoted as d_G.

We assume all intermediate results of our algorithm have word size $O(\log n)$ and basic algorithmic operations can be finished within a constant time. Then within the range of $[1, n^k]$, the integer part of natural logarithm of an integer and floor function of an real number can be computed in constant time for any constant k. This can be achieved using standard table-lookup techniques (similar approaches can be found in Thorup's algorithm [32]). The time complexity of the algorithms are measured using the random-access machine (RAM) model.

A result holds *with high probability* (**w.h.p.**) for an input of size n if it holds with probability at least $1 - n^{-c}$ for any constant $c > 0$, over all possible random choices made by the algorithm.

Let $[n] = \{1, 2, \cdots, n\}$ where n is a positive integer.

We recall a useful fact about random permutations [31]:

▶ **Lemma 1.** *Let $\pi : [n] \to [n]$ be a permutation selected uniformly at random on $[n]$. The set $\{i \mid i \in [n], \pi(i) = \min\{\pi(j) \mid j = 1, \cdots, i\}\}$ contains $O(\log n)$ elements both in expectation and with high probability.*

3 An Approximate SSSP Algorithm

In this section we introduce a variant of Dijkstra's algorithm. This is an efficient algorithm for single-source shortest paths (SSSP) with linear time complexity $O(m)$. The computed distances are α-distance preserving:

▶ **Definition 2** (α-distance preserving)**.** For a weighted graph $G = (V, E)$, the single-source distances $d(v)$ for $v \in V$ from the source node s is α-distance preserving, if there exists a constant $0 \le \alpha \le 1$ such that $\alpha\, d_G(s, u) \le d(u) \le d_G(s, u)$, and $d(v) - d(u) \le d_G(u, v)$, for every $u, v \in V$.

α-distance preserving can be viewed as the triangle inequality on single-source distances (i.e. $d(u) + d_G(u, v) \ge d(v)$ for $u, v \in V$), and is required in many applications related to distances. For example, in Corollary 4 we show that using Gabow's scaling algorithm [20] we can compute a $(1 + \epsilon)$-approximate SSSP using $O(m \log \epsilon^{-1})$ time. Also in many metric problems including the contruction of optimal tree embeddings, distance preservation is necessary in the proof of the expected stretch, and such an example is Lemma 11 in Section 4.3.

The preliminary version we discussed in Section 3.1 limits edge weights in $[1, n^k]$ for a constant k, but with some further analysis in the full version of this paper we can extend the range to $[1, n^{O(m)}]$. This new algorithm also has two properties that are needed in the construction of FRT trees, while no previous algorithms achieve them all:

1. (α-distance preserving) The computed distances from the source $d(\cdot)$ is α-distance preserving.
2. (Ordering property and linear complexity) The vertices are visited in order of distance $d(\cdot)$, and the time to compute the first k distances is bounded by $O(m')$ where m' is the sum of degrees from these k vertices.

The algorithm also works on directed graphs, although this is not used in the FRT construction.

Approximate SSSP algorithms are well-studied [32, 23, 13, 29, 26]. In the sequential setting, Thorup's algorithm [32] compute single-source distances on undirected graphs with integer weights using $O(n+m)$ time. Nevertheless, Thorup's algorithm does not obey the ordering property since it uses a hierarchical bucketing structure and does not visit vertices in an order of increasing distances, and yet we are unaware of a simple argument to fix this. Other algorithms are either not work-efficient (i.e. super-linear complexity) in sequential setting, and / or violating distance preservation.

▶ **Theorem 3.** *For a weighted directed graph $G=(V,E)$ with edge weights between* 1 *and* $n^{O(1)}$*, a* $(1/4)$*-distance preserving single-source shortest-path distances* $d(\cdot)$ *can be computed, such that the distance to the* k*-nearest vertices* v_1 *to* v_k *by* $d(\cdot)$*requires* $O(\sum_{i=1}^{k}\mathrm{degree}(v_i))$ *time.*

The algorithm also has the two following properties. Since they are not used in the construction of FRT trees, we review them in the full version of this paper. We discuss how to (1) extend the range of edge weights to $n^{O(m)}$, and the cost to compute the k-nearest vertices is $O(\log_n d(v_k)+\sum_{i=1}^{k} \mathit{degree}(v_i))$ where v_1 to v_k are the k nearest vertices; and (2) compute $(1+\epsilon)$-distance-preserving shortest-paths for an arbitrary $\epsilon>0$:

▶ **Corollary 4.** $(1+\epsilon)$*-distance-preserving shortest-paths for all vertices can be computed by repeatedly using Theorem 3* $O(\log \epsilon^{-1})$ *times.*

3.1 Algorithm Details

The key data structure in this algorithm is a bucket-tree structure shown in Figure 1 that implements the priority queue in Dijkstra's algorithm. With the bucket-tree structure, each Decrease-Key or Extract-Min operation takes constant time. Given the edge range in $[1,n^k]$, this structure has $l=\lceil(1+k)\log_2 n\rceil$ levels, each level containing a number of buckets corresponding to the distances to the source node. In the lowest level (level 1) the difference between two adjacent buckets is 2.

At anytime only one of the buckets in each level can be non-empty: there are in total l active buckets to hold vertices, one in each level. The active bucket in each level is the left-most bucket whose distance is larger than that of the current vertex being visited in our algorithm. We call these active buckets the **frontier** of the current distance, and they can be computed by the **path string**, which is a 0/1 bit string corresponding to the path from the current location to higher levels (until the root), and 0 or 1 is decided by whether the node is the left or the right child of its parent. For clarity, we call the buckets on the frontier **frontier buckets**, and the ancestors of the current bucket **ancestor buckets** (can be traced using the path string). For example, as the figure shows, if the current distance is 4, then the available buckets in the first several levels are the buckets corresponding to the distances $6,5,11,7$, and so on. The ancestor bucket and the frontier bucket in the same level may or may not be the same, depending on whether the current bucket is the left or right subtree of this bucket. For example, the path string for the current bucket with label 4 is 0100 and so on, and ancestor buckets correspond to $4,5,3,7$ and so on. It is easy to see that given the current distance, the path string, the ancestor buckets, and the frontier buckets can be computed in $O(l)$ time—constant time per level.

Note that since only one bucket in each level is non-empty, the whole structure need not to be build explicitly: we store one linked list for each level to represent the only active bucket in the memory (l lists in total), and use the current distance and path string to retrieve the location of the current bucket in the structure.

With the bucket-tree structure acting as the priority queue, we can run standard Dijkstra's algorithm. The only difference is that, to achieve linear cost for an SSSP query, the operations of Decrease-Key and Extract-Min need to be redefined on the bucket-tree structure.

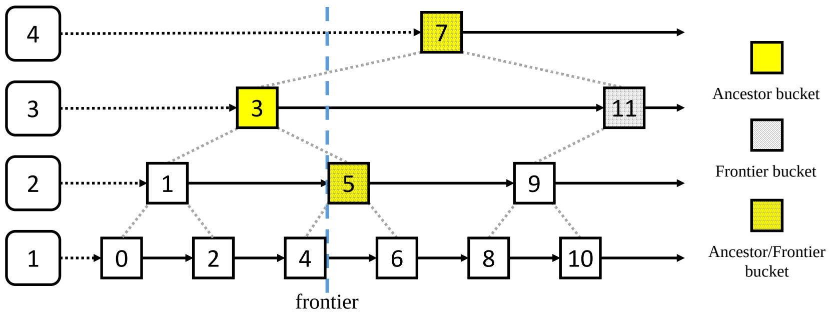


Figure 1 An illustration of a bucket-tree structure with the lowest 4 levels, and the current visiting bucket has distance 4. Notice that our algorithm does not insert vertices to the same level as the current bucket (i.e. bucket 6).

Once the relaxation of an edge succeeds, a Decrease-Key operation for the corresponding vertex will be applied. In the bucket-tree structure it is implemented by a Delete (if the vertex is added before) followed by an Insert on two frontier buckets respectively. The deletion is trivial with a constant cost, since we can maintain the pointer from each vertex to its current location in the bucket tree. We mainly discuss how to insert a new tentative distance into the bucket tree. When vertex u successfully relaxes vertex v with an edge e, we first round down the edge weight w_e by computing $r = \lfloor \log_2 (w_e + 1) \rfloor$. Then we find the appropriate frontier bucket B that the difference of the distances w'_e between this bucket B and the current bucket is the closest to (but no more than) $w_r = 2^r - 1$, and insert the relaxed vertex into this bucket. The constant approximation for this insertion operation holds due to the following lemma:

▶ **Lemma 5.** *For an edge with length w_e, the approximated length w'_e, which is the distance between the inserted bucket B and the current bucket, satisfies the following inequality: $w_e/4 \le w'_e \le w_e$.*

Proof. After the rounding, $w_r = 2^r - 1 = 2^{\lfloor \log_2 (w_e+1) \rfloor} - 1$ falls into the range of $[w_e/2, w_e]$. We now show that there always exists such a bucket B on the frontier that the approximated length w'_e is in $[w_r/2, w_r]$.

We use Algorithm 1 to select the appropriate bucket for a certain edge, given the current bucket level and the path string. The first case is when b, the current level, is larger than r. In this case all the frontier buckets on the bottom r levels form a left spine of the corresponding subtree rooted by the right child of the current bucket, so picking bucket in the r-th level leads to $w'_e = 2^{r-1}$, and therefore $w_e/4 < w'_e \le w_e$ holds. The second case is when $b \le r$, and the selected bucket is decided based on the structure on the ancestor buckets from the $(r+1)$-th level to $(r-1)$-th level, which is one of the three following cases.

- The simplest case ($b < r$, line 9) is when the ancestor bucket in the $(r-1)$-th level is the right child of the bucket in the r-th level. In this case when we pick the bucket in level r since the distance between two consecutive buckets in level r is 2^r, and the distance from the current bucket to the ancestor bucket in r-th level is at most $\sum_{i=1}^{r-1} 2^{i-1} < 2^{r-1}$. The distance thus between the current bucket and the frontier bucket in level r is $w'_e > 2^r - 2^{r-1} = 2^{r-1} > w_e/4$.
- The second case is when either $b = r$ and the current bucket is the left child (line 5), or $b < r$ and the ancestor bucket in level $r - 1$ is on the left spine of the subtree rooted at the ancestor bucket in level $r + 1$ (line 11). Similar to the first case, picking the frontier

Algorithm 1: Finding the appropriate bucket

```
Input: Current bucket level $b$, rounded edge length $2^r - 1$ and path string.
Output: The bucket in the frontier (the level is returned).
Let $r'$ be the lowest ancestor bucket above level $r$ that is a left child
if $b > r$ then
    return $r$
else if $b = r$ then
    if current bucket is left child then return $r + 1$
    else return $r' + 1$
else
    switch the branches from $(r + 1)$-th level to $(r - 1)$-th level in the path string do
        case left-then-right or right-then-right do
            return $r$
        case left-then-left do
            return $r + 1$
        case right-then-left do
            return $r' + 1$
```

bucket in the $(r + 1)$-th level (which is also an ancestor bucket) skips the right subtree of the bucket in r-th level, which contains $2^{r-1} - 1 \geq w_e/4$ nodes.

- The last case is the same as the second case expect that the level-r ancestor bucket is the right child of level-$(r + 1)$ ancestor bucket. In this case we will pick the frontier bucket that has distance 2^{r-1} to the ancestor bucket in level r, which is the parent of the lowest ancestor bucket that is a left child and above level r. In this case the approximated edge distance is between 2^{r-1} and $2^r - 1$.

Combining all these cases proves the lemma. ◀

We now explain ehe Extract-Min operation on the bucket tree. We will visit vertex in the current buckets one by one, so each Extract-Min has a constant cost. Once the traversal is finished, we need to find the next closest non-empty frontier.

▶ **Lemma 6.** *Extract-Min and Decrease-Key on the bucket tree require $O(1)$ time.*

Proof. We have shown that the modification on the linked list for each operation requires $O(1)$ time. A naïve implementation to find the bucket in Decrease-Key and Extract-Min takes $O(l) = O(\log n)$ time, by checking all possible frontier buckets. We can accelerate this look-up using the standard table-lookup technique. The available combinations of the input of Decrease-Key are n^{k+1} (total available current distance) by $l = O(k \log n)$ (total available edge distance after rounding), and the input combinations of Extract-Min are two $\lceil \log_2 n^{k+1} \rceil$ bit strings corresponding to the path to the root and the emptiness of the buckets on the frontier. We therefor partition the bucket tree into several parts, each containing $\lfloor (1 - \epsilon')(\log_2 n)/2 \rfloor$ consecutive levels (for any $0 < \epsilon' < 1$). We now precompute the answer for all possible combinations of path strings and edge lengths, and (1) the sizes of look-up tables for both operations to be $O((2^{\lfloor (1-\epsilon')(\log_2 n)/2 \rfloor})^2) = o(n)$, (2) the cost for brute-force preprocessing to be $O((2^{\lfloor (1-\epsilon')(\log_2 n)/2 \rfloor})^2 \log n) = o(n)$, and (3) the time of either operation of Decrease-Key and Extract-Min to be $O(k)$, since each operation requires to look up at most $l/\lfloor (1 - \epsilon')(\log_2 n)/2 \rfloor = O(k)$ tables. Since k is a constant, each of the two operations as well takes constant time. The update of path string can be computed similarly using this table-lookup approach. As a result, with $o(n)$ preprocessing time, finding the associated bucket for Decrease-Key or Extract-Min operation uses $O(1)$ time. ◀

We now show the three properties of the new algorithm: linear complexity, triangle inequality, and the ordering property.

Proof of Theorem 3. Here we show the algorithm satisfies the properties in Theorem 3. Lemma 6 proves the linear cost of the algorithm. Lemma 5 shows that the final distances is α-distance preserving. Lastly, since this algorithm is actually a variant of Dijkstra's algorithm with the priority implemented by the bucket-tree structure, the ordering property is met, although here the k-nearest vertices are based on the approximate distances instead of real distances. ◀

In the full version we discuss how to extend the range of edge weight to $[1, n^{O(m)}]$.

4 The Dominance Sequence

In this section we review and introduce the notion of dominance sequences for each point of a metric space and describe the algorithm for constructing them on a graph. The basic idea of dominance sequences was previously introduced in [12] and [8]. Here we name the structure as the dominance sequence since the "dominance" property introduced below is crucial and related to FRT construction. In the next section we show how they can easily be converted into an FRT tree.

4.1 Definition

▶ **Definition 7** (Dominance). Given a premetric (X, d_X) and a permutation π, for two points $x, y \in X$, x *dominates* y if and only if

$$\pi(x) = \min\{\pi(w) \mid w \in X, d_X(w, y) \leq d_X(x, y)\}.$$

Namely, x dominates y iff x's priority is greater (position in the permutation is earlier) than any point that is closer to y.

The dominance sequence for a point $x \in X$, is the sequence of all points that dominate x sorted by distance. More formally:

▶ **Definition 8** (Dominance Sequence[2]). For each $x \in X$ in a premetric (X, d_X), the *dominance sequence* of a point x with respect to a permutation $\pi : X \to [n]$ (denoted as $\chi_\pi^{(x)}$), is the sequence $\langle p_i \rangle_{i=1}^{k}$ such that $1 = \pi(p_1) < \pi(p_2) < \cdots < \pi(p_k) = \pi(x)$, and p_i is in $\chi_\pi^{(x)}$ iff p_i dominates x.

We use χ_π to refer to all dominance sequences for a premetric under permutation π. It is not hard to bound the size of the dominance sequence:

▶ **Lemma 9** ([13]). *Given a premetric (X, d_X) and a random permutation π, for each vertex $x \in X$, with w.h.p.*

$$\left|\chi_\pi^{(x)}\right| = O(\log n)$$

[2] Also called as "least-element list" in [12]. We rename it since in later sections we also consider many other variants of it based on the dominance property.

```
Algorithm 2: Efficient FRT tree construction
1 Pick a uniformly random permutation $\pi : V \to [n]$.
2 Compute the dominance sequences $\chi_\pi$.
3 Pick $\beta \in [1, 2]$ with the probability density function $f_B(x) = 1/(x \ln 2)$.
4 Convert the dominance sequence $\chi_\pi$ to the compressed partition sequence $\bar{\sigma}_{\pi,\beta}$.
5 Generate the FRT tree based on $\bar{\sigma}_{\pi,\beta}$.
```

and hence overall, with w.h.p.

$$|\chi_\pi| = \sum_{x \in X} \left|\chi_\pi^{(x)}\right| = O(n \log n)$$

Since the proof is fairly straight-forward, for completeness we also provide it in the full version of this paper.

Now consider a graph metric (V, d_G) defined by an undirected positively weighted graph $G = (V, E)$ with n vertices and m edges, and $d_G(u, v)$ is the shortest distance between u and v on G. The dominance sequences of this graph metric can be constructed using $O(m \log n + n \log^2 n)$ time w.h.p. [12]. This algorithm is based on Dijkstra's algorithm.

4.2 Efficient FRT tree construction based on the dominance sequences

We now consider the construction of FRT trees based on a pre-computed dominance sequences of a given metric space (X, d_X). We assume the weights are normalized so that $1 \le d_X(x, y) \le \Delta = 2^\delta$ for all $x \ne y$, where δ is a positive integer.

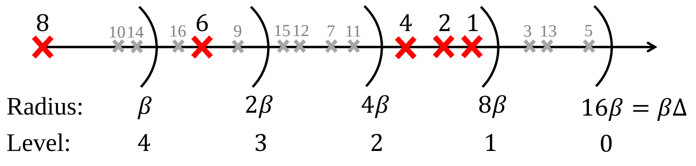
The FRT algorithm [17] generates a top-down recursive low-diameter decomposition (LDD) of the metric, which preserves the distances up to $O(\log n)$ in expectation. It first chooses a random β between 1 and 2, and generates $1 + \log_2 \Delta$ levels of partitions of the graph with radii $\{\beta\Delta, \beta\Delta/2, \beta\Delta/4, \cdots\}$. This procedure produces a laminar family of clusters, which are connected based on set-inclusion to generate the FRT tree. The weight of each tree edge on level i is $\beta\Delta/2^i$.

Instead of computing these partitions directly, we adopt the idea of a point-centric view proposed in [8]. We use the intermediate data structure "dominance sequences" as introduced in Section 4.1 to store the useful information for each point. Then, an FRT tree can be retrieved from this sequence with very low cost:

▶ **Lemma 10.** *Given β and the dominance sequences χ_π of a metric space with associated distances to all elements, an FRT tree can be constructed using $O(n \log n)$ time w.h.p.*

The difficulty in this process is that, since the FRT tree has $O(\log \Delta)$ levels and Δ can be large (i.e. $\Delta > 2^{O(n)}$), an explicit representation of the FRT tree can be very costly. Instead we generate the compressed version with nodes of degree two removed and their incident edge weights summed into a new edge. The algorithm is outlined in Algorithm 2.

Proof. We use the definition of partition sequence and compressed partition sequence from [8]. Given a permutation π and a parameter β, the *partition sequence* of a point $x \in X$, denoted by $\sigma_{\pi,\beta}^{(x)}$, is the sequence $\sigma_{\pi,\beta}^{(x)}(i) = \min\{\pi(y) \mid y \in X, d(x, y) \le \beta \cdot 2^{\delta-i}\}$ for $i = 0, \ldots, \delta$, i.e. point y has the highest priority among vertices up to level i. We note that a trie (radix tree) built on the partition sequence is the FRT tree, but as mentioned we cannot build this explicitly. The compressed partition sequence, denoted as $\bar{\sigma}_{\pi,\beta}^{(x)}$, replaces consecutive



Dominance sequence:

$\chi_\pi^{(8)} = \langle 1,2,4,6,8 \rangle$

Partition sequence:

$\sigma_{\pi,\beta}^{(8)} = \langle 1,1,6,6,8 \rangle$

Compressed partition sequence:

$\bar{\sigma}_{\pi,\beta}^{(8)} = \langle (1,0),(6,2),(8,4) \rangle$

Figure 2 An illustration for dominance sequence, partition sequence and compressed partition sequence for vertex 8. Here we assume that the label of each vertex corresponds to its priority. The left part shows the distances of all vertices to vertex 8 in log-scale, and the red vertices dominate vertex 8.

equal points in the partition sequence $\sigma_{\pi,\beta}^{(x)}$ by the pair (p_i, l_i) where p_i is the vertex and l_i is the highest level p_i dominates x in the FRT tree. Figure 2 gives an example of a partition sequence, a compressed partition sequence, and their relationship to the dominance sequence.

To convert the dominance sequences χ_π to the compressed partition sequences $\bar{\sigma}_{\pi,\beta}$ note that for each point x the points in $\bar{\sigma}_{\pi,\beta}^{(x)}$ are a subsequence of $\chi_\pi^{(x)}$. Therefore, for $\bar{\sigma}_{\pi,\beta}^{(x)}$, we only keep the highest priority vertex in each level from $\chi_\pi^{(x)}$ and tag it with the appropriate level. Since there are only $O(\log n)$ vertices in $\chi_\pi^{(x)}$ w.h.p., the time to generate $\bar{\sigma}_{\pi,\beta}^{(x)}$ is $O(\log n)$ w.h.p., and hence the overall construction time is $O(n \log n)$ w.h.p.

The compressed FRT tree can be easily generated from the compressed partition sequences $\bar{\sigma}_{\pi,\beta}$. Blelloch et. al. [8] describe a parallel algorithm that runs in $O(n^2)$ time (sufficient for their purposes) and polylogarithmic depth. Here we describe a similar version to generate the FRT tree sequentially in $O(n \log n)$ time w.h.p. The idea is to maintain the FRT as a patricia trie [27] (compressed trie) and insert the compressed partition sequences one at a time. Each insertion just needs to follow the path down the tree until it diverges, and then either split an edge and create a new node, or create a new child for an existing node. Note that a hash table is required to trace the tree nodes since the trie has a non-constant alphabet. Each insertion takes time at most the sum of the depth of the tree and the length of the sequence, giving the stated bounds. ◀

We note that for the same permutation π and radius parameter β, it generates exactly the same tree as the original algorithm in [17].

4.3 Expected Stretch Bound

In Section 4.2 we discussed the algorithm to convert the dominance sequences to a FRT tree. When the dominance sequences is generated from a graph metric (G, d_G), the expected stretch is $O(\log n)$, which is optimal, and the proof is given in many previous papers [17, 8]. Here we show that any distance function $\hat{d}_G$ in Lemma 11 is sufficient to preserve this expected stretch. As a result, we can use the approximate shortest-paths computed in Section 3 to generate the dominance sequences and further convert to optimal tree embeddings.

▶ **Lemma 11.** *Given a graph metric (G, d_G) and a distance function $\hat{d}_G(u,v)$ such that for $u, v, w \in V$, $|\hat{d}_G(u,v) - \hat{d}_G(u,w)| \le 1/\alpha \cdot d_G(v,w)$ and $d_G(u,v) \le \hat{d}_G(u,v) \le 1/\alpha \cdot d_G(u,v)$ for some constant $0 < \alpha \le 1$, then the dominance sequences based on $(G, \hat{d}_G)$ can still yield optimal tree embeddings.*

Proof Outline. Since the overestimate distances hold the dominating property of the tree embeddings, we show the expected stretch is also not affected. We now show the expected stretch is also held.

Recall the proof of the expected stretch by Blelloch et al. in [8] (Lemma 3.4). By replacing d_G by $\hat{d}_G$, the rest of the proof remains unchanged except for Claim 3.5, which upper bounds the expected cost of a common ancestor w of $u, v \in V$ in u and v's dominance sequences. The original claim indicates that the probability that u and v diverges in a certain level centered at vertex w is $O(|d_G(w,u) - d_G(w,v)|/d_G(u,w)) = O(d_G(u,v)/d_G(u,w))$ and the penalty is $O(d_G(u,w))$, and therefore the contribution of the expected stretch caused by w is the product of the two, which is $O(d_G(u,v))$ (since there are at most $O(\log n)$ of such w (Lemma 9), the expected stretch is thus $O(\log n)$). With the distance function $\hat{d}_G$ and α as a constant, the probability now becomes $O(|\hat{d}_G(w,u) - \hat{d}_G(w,v)|/\hat{d}_G(u,w)) = O(d_G(u,v)/d_G(u,w))$, and the penalty is $O(\hat{d}_G(u,w)) = O(d_G(u,w))$. As a result, the expected stretch asymptotically remains unchanged. ◀

4.4 Efficient construction of approximate dominance sequences

Assume that $\hat{d}_G(u,v)$ is computed as $d_u(v)$ by the shortest-path algorithm in Section 3 from the source node u. Notice that $d_u(v)$ does not necessarily to be the same as $d_v(u)$, so $(G, \hat{d}_G(u,v))$ is not a metric space. Since the computed distances are distance preserving, it is easy to check that Lemma 11 is satisfied, which indicate that we can generate optimal tree embeddings based on the distances. This leads to the main theorem of this paper.

▶ **Theorem 12** (Efficient optimal tree embeddings). *There is a randomized algorithm that takes an undirected positively weighted graph $G = (V, E)$ containing $n = |V|$ vertices and $m = |E|$ edges, and produces an tree embedding such that for all $u, v \in V$, $d_G(u,v) \le d_T(u,v)$ and $\mathbf{E}[d_T(u,v)] \le O(\log n) \cdot d_G(u,v)$. The algorithm w.h.p. runs in $O(m \log n)$ time.*

The algorithm computes approximate dominance sequences $\hat{\chi}_\pi$ by the approximate SSSP algorithm introduced in Section 3. Then we apply Lemma 10 to convert $\hat{\chi}_\pi$ to an tree embedding. Notice that this tree embedding is an FRT-embedding based on $\hat{d}_G$. We still call this an FRT-embedding since the overall framework to generate hierarchical tree structure is similar to that in the original paper [17].

The advantage of our new SSSP algorithm is that the Decrease-Key and Extract-Min operation only takes constant time when the relative edge range (maximum divided by minimum) is no more than $n^{O(1)}$. To handle arbitrary weight edges we adopt a similar idea to [23] to solve the subproblems on specific edge ranges, and concatenate the results to form the final output. This requires pre-processing to restrict the edge range.

The high-level idea of the algorithm is as follows. In the pre-processing, the goal is to use $O(m \log n)$ time to generate a list of subproblems: the i-th subproblem has edge range in the interval $[n^{i-1}, n^{i+1}]$ and we compute the elements in the dominance sequences with values falling into the range from n^i to n^{i+1}. All the edge weights less than the minimum value of this range are treated as 0. Namely, the vertices form some components in one subproblem and the vertex distances within each component is 0.

After the subproblems are computed, we run the shortest-path algorithm on each subproblem, and the i-th subproblem generates the entries in the approximate dominance sequences in the range of $[n^i, n^{i+1})$. Finally, we solve an extra subproblem that only contains edges with weight less than n to generate the elements in dominance sequences whose distances fall in range $[1, n)$.

Due to page limit, the details of the algorithm, the proofs of the correctness and time bound are given in the full version of this paper.

5 An Application of FRT-Embedding: Ramsey Partitions and Distance Oracles

In this section we show a new application of FRT-embedding, which with our efficient construction, accelerates the construction of some existing approximate distance oracles [34, 10]. The bridge is to show that the FRT trees are Ramsey partitions [24]. It is interesting to point out that, the construction of FRT trees is not only much faster and simpler than the previous best-known approach [25] to generate Ramsey partitions, but the stretch factor of k, which is 18.5, is also smaller than the previous constants of 128 [24] and 33 [28].

We start with the definition of Ramsey partitions. Let (X, d_X) be a metric space. A hierarchical partition tree of X is a sequence of partitions $\{\mathcal{P}_k\}_{k=0}^{\infty}$ of X such that $\mathcal{P}_0 = \{X\}$, the diameter of the partitions in each level decreases by a constant $c > 1$, and each level $\mathcal{P}_{k+1}$ is a refinement of the previous level $\mathcal{P}_k$. A Ramsey partition [24] is a distribution of hierarchical partition trees such that each vertex has a lower-bounded probability of being sufficiently far from the partition boundaries in all partitions k, and this gap is called the *padded range* of a vertex. More formally:

▶ **Definition 13.** An (α, γ)-Ramsey partition of a metric space (X, d_X) is a probability distribution over hierarchical partition trees $\mathcal{P}$ of X such that for every $x \in X$:

$$\Pr\left[\forall k \in \mathbb{N}, B_X\left(x, \alpha \cdot c^{-k}\Delta\right) \subseteq \mathcal{P}_k(x)\right] \geq |X|^{-\gamma}.$$

An asymptotically tight construction of Ramsey partition where $\alpha = \Omega(\gamma)$ is provided by Mendel and Naor [24] using the Calinescu-Karloff-Rabani partition [9] for each level.

▶ **Theorem 14.** *The probability distribution over FRT trees is an asymptotically tight Ramsey Partition with $\alpha = \Omega(\gamma)$ (shown in the appendix) with fixed $c = 2$. More precisely, for every $x \in X$,*

$$\Pr\left[\forall i \in \mathbb{N}, B_X\left(x, \left(1 - 2^{-1/2a}\right)2^{-i}\Delta\right) \subseteq \mathcal{P}_i(x)\right] \geq \frac{1}{2}|X|^{-\frac{2}{a}}$$

for any positive integer $a > 1$.

The details of the proof are provided in the full version of this paper.

Ramsey Partitions are used in generating approximate distance oracles (ADOs), which supports efficient approximate pairwise shortest-distance queries. ADOs are well-studied by many researchers (e.g. [33, 5, 1, 16, 11, 24, 34, 10]), and a (P, S, Q, D)-distance oracle on a finite metric space (X, d_X) is a data structure that takes expected time P to preprocess from the given metric space, uses S storage space, and answers distance query between points x and y in X in time Q satisfying $d_X(x, y) \leq d_O(x, y) \leq D \cdot d_X(x, y)$, where $d_O(x, y)$ is the pairwise distance provided by the distance oracle, and D is called the stretch.

▶ **Corollary 15.** *With Theorem 14, we can accelerate the time to construct the Ramsey-partitions-based Approximate Distance Oracle in [24] to*

$$O\left(n^{1/k}(m + n \log n) \log n\right)$$

on a graph with n vertices and m edges, improving the stretch to $18.5k$, while maintaining the same storage space and constant query time.

This can be achieved by replacing the original hierarchical partition trees in the distance oracles by FRT trees (and some other trivial changes). The construction time can further reduce to $O\big(n^{1/k}m\log n\big)$ using the algorithm introduced in Section 4.4 while the oracle still has a constant stretch factor. Accordingly, the complexity to construct Christian Wulff-Nilsen's Distance Oracles [34] and Shiri Chechik's Distance Oracles [10] can be reduced to

$$O\Big(kmn^{1/k} + kn^{1+1/k}\log n + n^{1/ck}m\log n\Big)$$

since they all use Mendel and Naor's Distance Oracle to obtain an initial distance estimation. The acceleration is from two places: first, the FRT tree construction is faster; second, FRT trees provide better approximation bound, so the c in the exponent becomes smaller.

References

1 Rachit Agarwal and Philip Godfrey. Distance oracles for stretch less than 2. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 526–538, 2013.

2 Noga Alon, Richard M. Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1):78–100, 1995.

3 Yair Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of IEEE Foundations of Computer Science (FOCS)*, pages 184–193, 1996.

4 Yair Bartal. On approximating arbitrary metrices by tree metrics. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 161–168. ACM, 1998.

5 Surender Baswana and Telikepalli Kavitha. Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 591–602, 2006.

6 Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 467–478, 2016.

7 Guy E. Blelloch, Yan Gu, and Yihan Sun. Efficient construction of probabilistic tree embeddings. *arXiv preprint arXiv:1605.04651*, 2016. URL: `https://arxiv.org/abs/1605.04651`.

8 Guy E. Blelloch, Anupam Gupta, and Kanat Tangwongsan. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *Proceedings of ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 205–213, 2012.

9 Gruia Calinescu, Howard Karloff, and Yuval Rabani. Approximation algorithms for the 0-extension problem. *SIAM Journal on Computing*, 34(2):358–372, 2005.

10 Shiri Chechik. Approximate distance oracles with constant query time. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 2014.

11 Shiri Chechik. Approximate distance oracles with improved bounds. In *Proceedings of ACM on Symposium on Theory of Computing (STOC)*, pages 1–10, 2015.

12 Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

13 Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 47(1):132–166, 2000.

14 Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving SDD linear systems in nearly $m\log^{1/2} n$ time. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 343–352, 2014.

15 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

16 Michael Elkin and Seth Pettie. A linear-size logarithmic stretch path-reporting distance oracle for general graphs. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 805–821, 2015.

17 Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *Journal of Computer and System Sciences (JCSS)*, 69(3):485–497, 2004.

18 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

19 Stephan Friedrichs and Christoph Lenzen. Parallel Metric Tree Embedding Based on an Algebraic View on Moore-Bellman-Ford. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 455–466, 2016.

20 Harold N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148–168, 1985.

21 Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Distributed Computing*, pages 197–211. Springer, 2014.

22 Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. *Distributed Computing*, 25(3):189–205, 2012.

23 Philip N. Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

24 Manor Mendel and Assaf Naor. Ramsey partitions and proximity data structures. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 109–118, 2006.

25 Manor Mendel and Chaya Schwob. Fast C-K-R partitions of sparse graphs. *Chicago Journal of Theoretical Computer Science*, pages 1–18, 2009. Article 2. `doi:10.4086/cjtcs.2009.002`.

26 Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 192–201, 2015.

27 Donald R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, October 1968.

28 Assaf Naor and Terence Tao. Scale-oblivious metric fragmentation and the nonlinear Dvoretzky theorem. *Israel Journal of Mathematics*, 192(1):489–504, 2012.

29 Seth Pettie and Vijaya Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, 34(6):1398–1431, 2005.

30 Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 255–264, 2008.

31 Raimund Seidel. *Backwards analysis of randomized geometric algorithms.* Springer, 1993.

32 Mikkel Thorup. Undirected single source shortest paths in linear time. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 12–21, 1997.

33 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.

34 Christian Wulff-Nilsen. Approximate distance oracles with improved query time. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 539–549, 2013.