Revisiting \(kd\)-tree for Nearest Neighbor Search

Parikshit Ram
p.ram@gatech.edu
IBM Research AI
Georgia, USA

Kaushik Sinha
kaushik.sinha@wichita.edu
Wichita State University
Kansas, USA

ABSTRACT
\(kd\)-tree [16] has long been deemed unsuitable for exact nearest-neighbor search in high dimensional data. The theoretical guarantees and the empirical performance of \(kd\)-tree do not show significant improvements over brute-force nearest-neighbor search in moderate to high dimensions. \(kd\)-tree has been used relatively more successfully for approximate search [36] but lack theoretical guarantees. In the article, we build upon randomized-partition trees [14] to propose \(kd\)-tree based approximate search schemes with \(O(d \log d + \log n)\) query time for data sets with \(n\) points in \(d\) dimensions and rigorous theoretical guarantees on the search accuracy. We empirically validate the search accuracy and the query time guarantees of our proposed schemes, demonstrating the significantly improved scaling for same level of accuracy.

KEYWORDS
nearest-neighbor search; similarity search; randomized algorithms; space-partitioning trees

ACM Reference Format:

1 MAKING \(kd\)-TREE COMPETITIVE
We focus on the ubiquitous and well-studied problem of Euclidean nearest-neighbor search – for any set of points \(S \subset \mathbb{R}^d\) and any query \(q \in \mathbb{R}^d\), find the point in \(S\) closest to \(q\) with respect to the \(L_2\) metric. The brute-force solution of scanning the complete set \(S\) for a single \(q\) becomes infeasible for sets with large cardinality (that is, \(|S| = n\)). \(kd\)-tree [16] was one of the first space-partitioning tree indices proposed to solve this problem exactly in logarithmic time (that is, in \(O(\log n)\) time) using a depth-first tree traversal algorithm followed by backtracking. The \(kd\)-tree is an axis-aligned partition of the space containing the set \(S\), resulting in a hierarchical index of hyper-rectangles. However, the \(kd\)-tree based nearest-neighbor search suffered from the “curse of dimensionality” resulting in performance (empirical and theoretical) equivalent to or worse than the brute-force solution in many cases. Subsequently, various other space-partitioning trees (with related tree-traversal algorithms) such as metric trees [35], cover trees [10], PCA-trees [34] and more were explored for exact and approximate nearest-neighbor search.

Recognizing the hardness of exact nearest-neighbor search, especially in high dimensions, the focus shifted to finding the approximate nearest-neighbors – neighbors which are approximately as close as exact nearest-neighbors. Locality sensitive hashing (LSH) is one of the most popular techniques for approximate nearest-neighbor search [24], with various variants and improvements. The methods take advantage of randomization to provide worst-case guarantees for search accuracy and/or query time. An alternate to LSH are data-dependent schemes such as product quantization [25] and similarity graphs [22], which have been shown to be extremely efficient empirically. However, these data dependent schemes lack rigorous theoretical guarantees, making it hard to understand how well these schemes work and when these schemes might fail. \(kd\)-tree has also been shown to be empirically competitive for approximate nearest-neighbor search with the introduction of randomization and ensembling, resulting in a widely popular FLANN nearest-neighbor search tool [36]. However, this incarnation of \(kd\)-tree still lacks favorable theoretical guarantees for high dimensional data.

Relatively recently, randomized-partition tree (RPTree) [14] was proposed for nearest-neighbor search, with theoretical guarantees on the search accuracy for the \(O(d \log n)\) defeatist-tree search algorithm (depth-first tree traversal without any backtracking), and an ensemble of RPTrees were shown to empirically outperform LSH [40] – it is hard to compare RPTree and LSH theoretically since their guarantees have different forms (detailed in Section 2).

In this paper, we demonstrate how the theoretical search accuracy guarantees of a RPTree can be transferred to a \(kd\)-tree with improved search time scaling. Specifically, we show that:

- A \(kd\)-tree built on a randomly rotated version of the data set \(S\) has the same search accuracy guarantees as a RPTree, leading to a \(O(d^2 + \log n)\) search algorithm based on a \(kd\)-tree.
- A \(O(d \log d)\) approximate random rotation of the data allows \(kd\)-tree to retain the above search accuracy guarantees, resulting in a \(O(d \log d + \log n)\) \(kd\)-tree based search algorithm – an improvement over the \(O(d \log n)\) search time of a RPTree while possessing the same search accuracy guarantees.

These results allow a \(kd\)-tree based nearest-neighbor search scheme with rigorous theoretical guarantees on the search accuracy and a time complexity almost linear in \(d\) and logarithmic in \(n\).

We discuss the existing literature on nearest-neighbor search and the relevant background for our proposed schemes in Section 2. We present the new \(kd\)-tree based search schemes with their theoretical guarantees in Section 3 and present relevant empirical evaluation.
of the proposed schemes in Section 4. Finally, we conclude with the limitations and the potential future improvements in Section 5.

2 NEAREST-NEIGHBOR SEARCH

Nearest-neighbor search or similarity search is commonly encountered in computer science (for example, in machine learning, data mining) as well as in other physical sciences. Assuming that the items in the set $S$ being searched over have some numeric representation, the set $S$ of $n$ items is encoded as a subset of $\mathbb{R}^d$. The problem of exact nearest-neighbor search for any query $q \in \mathbb{R}^d$ with respect to a distance (or dissimilarity) function $d : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_+$ is to find the point $p^* \in S$ such that

$$p^* = \arg\min_{p \in S} d(q, p). \quad (1)$$

The most widely studied version of this problem focuses on the $\ell_2$ metric as the distance function $d(q, p) = \|q - p\|_2$. Nearest-neighbor search with general normed metrics are often reduced to the $\ell_2$ metric [3, 4] for efficient (approximate) solution. Space-partitioning tree indices (such as $kd$-tree [16], metric tree [35]) provided fast exact solutions for low to moderately high dimensions. However, their performances degrade in higher dimensions (higher values of $d$). One critical reason for the unfavourable performance of these tree-based exact search schemes is that even though the algorithm is able to find the nearest-neighbor quite fast, the process of empirically certifying that the candidate neighbor is in fact the nearest is computationally very expensive [39].

With the focus on approximate nearest-neighbor search, the usual procedure involves indexing the set $S$. For any query $q$, the index (in conjunction with an index traversal algorithm) is used to return a subset $S_q \subseteq S$, and the brute-force algorithm is used over $S_q$ to find the approximate nearest-neighbor. The query time has two major parts: (i) the time to process $q$ through the index to obtain the candidate neighbors, and (ii) the time to search over the candidates (possibly brute-force) to find the best candidate for $q$. The quality of the search is usually quantified as the recall of the query’s true nearest-neighborhood. The precision is directly related to the search efficiency – high precision implies a small (and high-quality) subset $S_q$, which means the brute-force algorithm processes $S_q$ faster. The overall success of any index-based search scheme is judged by its precision-recall trade-off (if available). Some common and representative search schemes are:

(i) The construction of binary-space-partitioning trees [5, 12, 16, 30, 36] on the set $S$ in conjunction with a tree-traversal algorithm (such as a depth-first defeatist search algorithm) for any query $q$ to obtain candidate neighbors.
(ii) Locality preserving hashing of $S$ and $q$ into a table [11, 15, 19, 24], and utilizing multiple hash tables to boost recall.
(iii) The generation of a codebook which quantizes the points in $S$ in a data-dependent manner such that nearby points are quantized to the same code [18, 25, 48], and similar codes indicate points in relatively close proximity.
(iv) The generation of a (sparse) similarity graph between the points in $S$ [22, 32, 33, 49] followed by a greedy graph traversal algorithm; these bear similarities to the traditional skip-lists

1. If $S_q^*$ is the true neighbor set of $q$, recall is defined as $|S_q^* \cap S_q|/|S_q^*|$.
2. The precision term is defined as $|S_q \cap S_q^*|/|S_q|$.

(see Ken Clarkson’s survey [12]) used in conjunction with Orchard’s algorithm [38] or AESA [45]. The hashing based schemes are extremely popular. They have favorable theoretical guarantees and empirical performance. The quantization based methods and the similarity graph based methods are currently considered the state-of-the-art in terms of empirical performance. However, they lack any theoretical guarantees. FLANN [36] demonstrated that $kd$-tree based approximate search can also be extremely competitive even in high-dimensional data (FLANN actually uses both $kd$-tree and $k$-means tree [17, 37]). However, FLANN lacks any theoretical guarantees. The results we present here can be considered as a step towards filling that gap.

2.1 Randomized partition trees for search

RPTree is a binary space-partitioning tree which comes with favorable theoretical guarantees [14] and is empirically shown to outperform locality-sensitive hashing based schemes [40]. Whether RPTree avoids the curse of dimensionality depends on the underlying structure of the problem. We cannot directly compare the theoretical guarantees of RPTree and LSH: With LSH, the guarantees are of the following form for the planted nearest-neighbor problem by solving the point-location in Euclidean ball problem – for a given $\varepsilon > 0$, the search returns all points within a distance of $(1+\varepsilon)\times$ the true nearest-neighbor distance with high probability in time $O(n^p)$ where $p < 1$, while being polynomial in the ambient dimensionality $d$. There are a couple of follow-up comments: (i) The guarantee requires knowledge of the true nearest-neighbor distance. In practice, this is handled by guess-and-halve. (ii) This requirement imposes practical issues where some queries have small or empty candidate sets, while others get large candidate sets, making it hard to control the precision-recall trade-off.

There has been numerous improvements to LSH based search over the last two decades. A lot of focus has been on learning data-dependent hash functions [20, 46, 47] such that the search returns better quality results compared to vanilla LSH. The schemes focus on learning data-structures/hashtables (implicitly learning representations) on the data that preserve local neighborhoods while separating non-neighbors with lowest possible number of hash bits.

On the other hand, a RPTree [14] provides guarantees of the following form – for a query $q$ and a set $S$ of $n$ points in $\mathbb{R}^d$, with a search time $O(d \log n)$, the search does not return the exact nearest-neighbor with probability at most $\lambda$ where $\lambda = \Phi(q, S)$ where

$$\Phi(q, S) = \frac{1}{n-1} \sum_{i=2}^{n} \|q - p_i\|_2^2,$$

and $p_i$ is the $i^{th}$ nearest-neighbor of $q$ in $S$. $\Phi$ is called the potential function. This definition is also extended to the $k$-nearest-neighbor problem for $k > 1$ (Theorem 9(b) in [14]) where the probability that the recall of the $k$ neighbors is not equal to 1 is upper-bounded.

There are a couple of things to note here:

- A small potential function value indicates that the nearest-neighbor is significantly closer to $q$ than the rest of the points

1 Although this is practically handled via multi-probe LSH [31].
2 $\Phi(q)$ can be explicitly computed, albeit, at $O(dn)$ cost per $q$. However, this quantity does not need to be computed for the purposes of the search.
in $S$. A large value indicates that all items (including the nearest-neighbor) are almost equally far away from $q$.

- The probability of missing the exact nearest-neighbor can be made arbitrarily small by utilizing multiple RPTrees. Figure 1 illustrates this search procedure. Using $L$ trees boosts the success probability to $1 - \lambda^L$ with a $O(Ld \log n)$ search time.
- A forest of RPTrees allows for fine-grained control over the precision-recall tradeoff $- L$ (approximately) balanced trees with maximum leaf-size $n_0$ always returns at most $Ln_0$ candidates, and users can control the trade-off by varying $L$.
- A crucial distinction is that RPTree provides a high probability guarantee on the exact nearest-neighbor while guarantees for LSH are only on the approximate nearest-neighbor.

However, the original RPTree has $O(nd)$ space complexity which was subsequently reduced to $O(nd^\rho)$ for some $\rho \in (0, 1)$ [41] and $O(d \log n)$ [28] using different techniques. A variant of RPTree [23] was also shown to be empirically very efficient although the precise search scheme lacks any theoretical guarantees. For $L$ trees with leaf size $n_0$, the overall search time for RPTree based scheme is naively $O(Ld \log(n/n_0) + Ldn_0)$. For a small constant $n_0$, the search scales with $O(Ld \log n)$ (See appendix Section A for a detailed discussion on the choice of $n_0$). In practice, the $Ldn_0$ term can be significantly reduced by using techniques like voting [23], partial optimized distance computation and even quantization, while still retaining the theoretical accuracy guarantees of RPTree. Another important distinction is that the widely used SimHash [11] and the more recent near-optimal LSH [2] solve (and provide guarantees) for the angular nearest-neighbor search, which is equivalent to $\ell_2$ nearest-neighbor search if all the points in $S$ have the same norm. In practice, these are useful for $\ell_2$ nearest-neighbor search since points close in $\ell_2$

are close in angle; the only issue is that the search might return a lot of false positive candidates, hurting the precision without affecting the recall. In contrast, the guarantees for RPTree are on the $\ell_2$ nearest-neighbor problem.

A connection was made between random-projection tree [13] and $kd$-tree by Vempala, 2012 [43] – it was shown that a $kd$-tree built on a data set after a randomized rotation adapts to the intrinsic dimensionality (specifically, the Assouad dimension [6]). A binary space partitioning tree is said to adapt to the intrinsic dimensionality $d' < d$ if the diameter of any node in the tree halves after $O(d')$ splits. The result was in a flavor similar to the intrinsic dimension adaptivity of the random-projection trees proposed by Dasgupta & Freund, 2008 [13] while preserving the axis-aligned splits of the $kd$-tree. The adaptivity guarantees of the random-projection tree was in terms of a notion of intrinsic dimensionality known as the local covariance dimension. Neither Vempala, 2012 [43] nor Dasgupta & Freund, 2008 [13] explicitly focused on nearest-neighbor search. The general understanding was that better adaptivity to the intrinsic dimension leads to better search performance, but an explicit (theoretical) connection was never made [44].

The RPTree (randomized partition tree) proposed by Dasgupta & Sinha, 2013 [14] for search is very similar to the aforementioned random-projection tree proposed by Dasgupta & Freund, 2008 [13] – there are subtle differences, mostly because they are geared towards different explicit goals. In the following section, we present a $kd$-tree based search scheme that retains the search accuracy guarantees of RPTree while achieving an improved search time scaling.
Rule 10

Make leaf node

return $T$

end

Function SampleNode: $(S, n_0, l) \rightarrow T$

if $|S| \leq n_0$ then
Make leaf node $T$ such that $S_T \leftarrow S$
elserule $\rho(\cdot) \leftarrow \text{ChooseRule}(S, l)$ // node split rule
$S_l \leftarrow \{x \in S : p(x) = 0\}$
$T_l \leftarrow \text{MakeTree}(S_l, n_0, l + 1)$ // left subtree
$S_r \leftarrow S \setminus S_l$
$T_r \leftarrow \text{MakeTree}(S_r, n_0, l + 1)$ // right subtree
$T \leftarrow (\rho(\cdot), T_l, T_r)$
return $T$
end

end

Algorithm 1: Tree indexing of set $S$ with preconditioner $P(\cdot)$ and the node splitting rule ChooseRule ($P(\cdot)$ and ChooseRule for different trees are presented in Table 1 and Algorithm 2).

Table 1: Preconditioners and node splitting rules for different trees (the variables used here are defined in the text; different choices for ChooseRule are detailed in Algorithm 2).

<table>
<thead>
<tr>
<th>Method</th>
<th>Preconditioner $P(x)$</th>
<th>ChooseRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPTree [14]</td>
<td>$x$</td>
<td>GaussianRP</td>
</tr>
<tr>
<td>SpGa: RPT [41]</td>
<td>$H_dD_x$</td>
<td>SparseGaussianRP</td>
</tr>
<tr>
<td>RR: KDTree</td>
<td>$F_x$</td>
<td>KDTreeSplit</td>
</tr>
<tr>
<td>RC: KDTree</td>
<td>$D_x \circ \gamma$</td>
<td>KDTreeSplit</td>
</tr>
<tr>
<td>FF: KDTree</td>
<td>$H_dG\Pi H_d^T D_x$</td>
<td>KDTreeSplit</td>
</tr>
</tbody>
</table>

2.2 RPTree construction and search algorithms

In this subsection, we present the precise algorithms for RPTree and our proposed schemes. The generic tree construction is presented in Algorithm 1 and the defeatist tree search is presented in Algorithm 3. The tree construction is preceded by a preconditioning step (Algorithm 1, line 3) where the data is modified in a pairwise-distance preserving way. Different space partitioning policies (tree node splitting rules ChooseRule) are presented in Algorithm 2 - GaussianRP function (line 2) corresponds to the original RPTree [14], SparseGaussianRP ($p$ routine (line 10) corresponds to a sparse RPTree (SpGa:RPT) with a sparsity parameter $p \in [0, 1]$ [41], and a simplified splitting policy of a kd-tree is presented in KDTreeSplit (line 19). The different choices for preconditioning are presented in Table 1.

Algorithm 2: Definitions of different node splitting rules ChooseRule used in the tree construction algorithm described in Algorithm 1 for a set $S$ at any tree level $l$ (for SparseGaussianRP, $p \in (0, 1)$ controls the sparsity level).

Node splitting rule $\rho(\cdot) \leftarrow \text{ChooseRule}(S, l)$

Function GaussianRP: $(S, l) \rightarrow \rho(\cdot)$ [14]

Sample random vector $w \in \mathbb{R}^d$: $w_i \sim N(0, 1)$
Sample diagonal matrix $B$ such $B_{ii} \sim B(p)$
Sample $eta \sim U(1/4, 3/4)$
$v \leftarrow \beta$-th fractile of $V$
$\rho(x) \leftarrow \mathbb{I}(w^Tv > v)$
return $\rho(\cdot)$

end

Function SparseGaussianRP ($p$: $(S, l) \rightarrow \rho(\cdot)$) [41]

Sample vector $w \in \mathbb{R}^d$: $w_i \sim N(0, 1)$
Sample diagonal matrix $B$ such $B_{ii} \sim B(p)$
Sample $eta \sim U(1/4, 3/4)$
$v \leftarrow \beta$-th fractile of $V$
$\rho(x) \leftarrow \mathbb{I}((Bw)^Tv > v)$
return $\rho(\cdot)$

end

Function KDTreeSplit: $(S, l) \rightarrow \rho(\cdot)$ [Section 3]

$i \leftarrow l \mod d$
$V \leftarrow \{x_i \forall x \in S\} // x_i$ $i$-th coordinate of $x$
Sample $eta \sim U(1/4, 3/4)$
$v \leftarrow \beta$-th fractile of $V$
$\rho(x) \leftarrow \mathbb{I}(x_i > v)$

end

$\mathbb{R}^d$ is the preconditioning function. $H_d \in \{-1, +1\}^{d \times d}$ is the Walsh–Hadamard matrix (see Equation (5)). $D$ is a diagonal matrix with $D_{ii} \sim \mathcal{R}$. $\Gamma \in \mathbb{R}^{d \times d}$ is a random rotation matrix with $\Gamma_{ij} \sim N(0, 1)$. $\circ$ is the circular convolution operator (see Equation (3)). $\gamma \in \mathbb{R}^d$ is a random vector with $\gamma_i \sim N(0, 1)$. $G$ is a diagonal matrix with $G_{ii} \sim N(0, 1)$. $\Pi \in \{0, 1\}^{d \times d}$ is a random permutation matrix.

3 RANDOM ROTATION + kd-TREE

Building upon Vempala, 2012 [43], we use the same scheme of randomly rotating the data set $S$ before building the kd-tree. Once $S$ is rotated via a multiplication to a random matrix $\Gamma$, the $kd$-tree is built using MakeTree (Algorithm 1), where ChooseRule is set as KDTreeSplit (Algorithm 2, line 19), giving us RR: KDTree (Table 1). Query processing requires a randomized rotation of $q$ followed by DefeatistTreeSearch (Algorithm 3) on the $kd$-tree with $\Gamma q$. This search procedure is illustrated in Figure 2. The randomized rotation takes $O(d^2)$ time, and the defeatist search with $kd$-tree takes $O(\log n)$ time, leading to a $O(d^2 + \log n)$ query time compared to the $O(d \log n)$ search time of the original RPTree.

Although we will subsequently show that RR: KDTree possesses the same theoretical guarantees on search accuracy as RPTree, the query time bound of $O(d^2 + \log n)$ quickly becomes prohibitive for moderately high dimensions because of the $d^2$ term. To circumvent
Algorithm 3:  Defeatist search algorithm for query \( q \) with tree \( T \) and preconditioner \( P(\cdot) \). This search algorithm is employed for all the binary space-partitioning trees considered here.

1. Candidate neighbor set \( S_q \leftarrow \text{SearchIndex}(q, P(\cdot), T) \)
2. Function SearchIndex: \((q, P(\cdot), T) \rightarrow S_q\)
3. \( S_q \leftarrow \text{DefeatistTreeSearch}(P(q), T) \)
4. return \( S_q \)
5. end
6. Function DefeatistTreeSearch: \((q, T) \rightarrow S_q\)
7. if \( T \) is a leaf node then
8. \( S_q \leftarrow S_T \) // all points in leaf node
9. else
10. \( \rho(\cdot), T_l, T_r \leftarrow T \) // Node split rule & children
11. if \( \rho(q) = 0 \) then // go left
12. \( S_q \leftarrow \text{DefeatistTreeSearch}(q, T_l) \)
13. else // go right
14. \( S_q \leftarrow \text{DefeatistTreeSearch}(q, T_r) \)
15. end
16. end
17. return \( S_q \)
18. end

this issue, we present two efficient ways of approximating the randomized rotation while retaining the guarantees of RPTree.

3.1 Randomized circular convolution + \( kd \)-tree

Circular convolution [21] between vectors \( x, y \in \mathbb{R}^d \) is defined as

\[
x \odot y = x^T \begin{bmatrix} y_1 & y_d & y_{d-1} & \cdots & y_2 \\ y_2 & y_1 & y_d & \cdots & y_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_d & y_{d-1} & y_{d-2} & \cdots & y_1 \end{bmatrix}.
\]

(3)

For a random Gaussian vector \( y \), \( x \odot y \) emulates a randomized rotation with the restriction that the rows of the rotation matrix are not independent. The main motivation behind using circular convolution is that Equation (3) can be efficiently computed as

\[
x \odot y = F^{-1}(F(x) \circ F(y)),
\]

(4)

where \( \circ \) is the Hadamard product (elementwise multiplication of vectors), and \( F(\cdot) \) is the Discrete Fourier Transform operator which can be computed in \( O(d \log d) \) time with Fast Fourier Transform (FFT) compared to the \( O(d^2) \) \( F \)x operation in \( RR:KDTree \). Circular convolution has previously been used in the context of nearest-neighbor search for learning binary embeddings [50].

For our purpose, we consider \( P(x) = Dx \odot y \) as the approximate randomized rotation operation of any point \( x \) (\( D \) and \( y \) are as defined in Section 2.2). We call this scheme \( RC:KDTree \). For any query \( q \), the \( P(q) = Dq \odot y \) can be performed in \( O(d \log d) \) time, with \( kd \)-tree DefeatistTreeSearch requiring \( O(\log n) \) time, giving us a final query time complexity of \( O(d \log d + \log n) \) for \( RC:KDTree \).

3.2 FastFood\([29]\) + \( kd \)-tree

Le et al., 2013 [29] build upon the fast Johnson Lindenstrauss transform [1] in the context of approximating expansions in kernel methods. Specifically, for diagonal matrices \( D \) and \( G \), random permutation matrix \( \Pi \) and Walsh-Hadamard matrix \( H_d \) (as defined in Section 2.2), they introduce the “FastFood” matrix \( H_dG\Pi H_dD \) as a surrogate for a dense Gaussian matrix and show that “any given row of the matrix HG\Pi HD is i.i.d. Gaussian” (Section 3.2 in [29]).

The matrix \( H_d \) is defined recursively with \( H_1 := 1 \):

\[
H_m = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{m/2} \\ -H_{m/2} \end{bmatrix}, \quad \forall m = 2^l, l \in \mathbb{Z}_+.
\]

(5)

This is assuming \( d = 2^l \) for some integer \( l > 0 \). This can be achieved in general by padding the points with additional columns of zeros.

For our purpose, we approximate the randomized rotation operation, a multiplication with a dense random matrix \( \Gamma \), with the multiplication to \( H_dG\Pi H_dD \). This gives us \( FF:KDTree \) approximate randomized rotation via FastFood followed by DefeatistTreeSearch (Algorithm 3) on the \( kd \)-tree. For any query \( q \), it can be shown that the \( H_dG\Pi H_dDq \) operation can be performed in \( O(d \log d) \) time, and the tree search takes \( O(\log n) \), giving us a query time of \( O(d \log d + \log n) \) similar to \( RC:KDTree \).

3.3 Properties of randomized \( kd \)-tree

In this subsection, we formally present the time complexity and search accuracy guarantees for the proposed schemes. For search accuracy, we show that the proposed schemes possess the same guarantees as the original RPTree. Theorem 1 demonstrates how the search accuracy guarantee of \( RR:KDTree \) is identical to that of RPTree, while Theorems 2 and 3 extend this result to provide similar guarantees for \( RC:KDTree \) and \( FF:KDTree \).

Theorem 1. Given any query \( q \in \mathbb{R}^d \), \( RR:KDTree \) requires \( O(d^2) \) preconditioning time, \( O(\log(n\|n_0\|)) \) tree traversal time, \( O(d\|n_0\|\) point processing time per tree and ensures that the failure probability of finding the exact nearest neighbor of \( q \) is identical to that of RPTree.

Proof. For the random rotation matrix \( \Gamma \) in any \( RR:KDTree \) \((\Pi, i.i.d. N(0, 1))\), let \( \Gamma_1, \ldots, \Gamma_d \in \mathbb{R}^d \) be the \( d \) rows of \( \Gamma \). For any \( x \in S \), the \( i^{th} \) coordinate of \( \Gamma x \in \mathbb{R}^d \) can be interpreted as projection of \( x \) onto a random projection direction \( \Gamma_i \), whose entries are i.i.d. \( N(0, 1) \). Therefore, at any internal node of \( RR:KDTree \), if the \( KTreeSplit \) function chooses the \( i^{th} \) coordinate of the randomly rotated \( S \) (via \( \Gamma \)), then during \( RR:KDTree \) construction, this internal node is split into left and right child nodes based the \( \beta \) fractile of the projected points corresponding to this internal node onto \( \Gamma_i \). From \( RPTree \) perspective, this can be thought of as storing a random projection direction \( \Gamma_i \) at this internal node and splitting of this node into left and right child nodes by projecting the points corresponding to this node onto \( \Gamma_i \) and based on \( \beta \) fractile \( v \) of this projected points. Similarly, while answering a query with \( DefeatistTreeSearch \), \( RR:KDTree \) routes the query \( q \) at this internal node (towards left or right child node) based on the \( i^{th} \) coordinate of \( \Gamma q \) by comparing it with the \( \beta \) fractile \( v \) of this node. Again, from \( RPTree \) perspective, this can be thought of as projecting \( q \) onto the stored random projection direction \( \Gamma_i \) at this node and comparing this projected value with \( v \). Therefore, the probability that \( q \) is separated from its nearest neighbor at any internal node of an
Table 2: Indexing and search costs of the different randomized partition trees (Precond. → Preconditioning, trav. → traversal, proc. → processing). The leaf size for each tree is at most $n_0$. For sparse RPTree, $p \in (0, 1)$ controls the projection vector sparsity.

<table>
<thead>
<tr>
<th>Method</th>
<th>Tree construction time</th>
<th>Index size</th>
<th>Precond. time</th>
<th>Tree trav. time</th>
<th>Leaf proc. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPTree</td>
<td>$(nd + n \log n)(\log(n/n_0))$</td>
<td>$d \log(n/n_0) + (n/n_0) + n$</td>
<td>–</td>
<td>$d \log(n/n_0)$</td>
<td>$dn_0$</td>
</tr>
<tr>
<td>SpGa:RPT</td>
<td>$nd \log d + (npd + n \log n)(\log(n/n_0))$</td>
<td>$d \log(n/n_0) + (n/n_0) + n$</td>
<td>$d \log d$</td>
<td>$pd \log(n/n_0)$</td>
<td>$dn_0$</td>
</tr>
<tr>
<td>SpRa:RPT</td>
<td>$nd \log d + (npd + n \log n)(\log(n/n_0))$</td>
<td>$d \log(n/n_0) + (n/n_0) + n$</td>
<td>$d \log d$</td>
<td>$pd \log(n/n_0)$</td>
<td>$dn_0$</td>
</tr>
<tr>
<td>RR:KDTree</td>
<td>$nd^2 + (n \log n)(\log(n/n_0))$</td>
<td>$d^2 + (n/n_0) + n$</td>
<td>$d^2$</td>
<td>$\log(n/n_0)$</td>
<td>$dn_0$</td>
</tr>
<tr>
<td>RC:KDTree</td>
<td>$nd \log d + (n \log n)(\log(n/n_0))$</td>
<td>$d + (n/n_0) + n$</td>
<td>$d \log d$</td>
<td>$\log(n/n_0)$</td>
<td>$dn_0$</td>
</tr>
<tr>
<td>FF:KDTree</td>
<td>$nd \log d + (n \log n)(\log(n/n_0))$</td>
<td>$d + (n/n_0) + n$</td>
<td>$d \log d$</td>
<td>$\log(n/n_0)$</td>
<td>$dn_0$</td>
</tr>
</tbody>
</table>

RPTree which represents the same set of points from $S$ and utilizes $\Gamma_1$ as random projection direction. The only difference between RPTree and RR:KDTree is that while the random projection direction at each internal node of an RPTree is different (thus requiring $O(n)$ distinct projection directions), in case of RR:KDTree only $d$ random projection directions are implicitly used. However, since the failure probability of finding the exact nearest-neighbor of $q$ by either RPTree or RR:KDTree is bounded from above by a union bound of the probabilities that $q$ is not separated from its nearest neighbor at any internal node along the query’s path from root to leaf, this probability bound is identical in both cases.

Finally, the preconditioning time is $O(d^2)$ for the matrix-vector multiplication $\Gamma q$: the tree traversal time is $O(\log(n/n_0))$ since the tree depth is $O(\log(n/n_0))$ and constant computation is required during query processing at any internal node along $q$’s path from root to leaf; $O(dn_0)$ time is required for the explicit distance computation between $q$ and the $n_0$ points in the leaf $q$ is routed to. □

**Theorem 2.** Given any $q \in \mathbb{R}^d$, RC:KDTree requires $O(d \log d)$ preconditioning time, $O(\log(n/n_0))$ tree traversal time, $O(dn_0)$ point processing time per tree and ensures that the failure probability of finding the exact nearest neighbor of $q$ is identical to that of RPTree.

**Proof.** Let $D$ and $y$ be as defined in Section 2.2. For any $x \in S$, the $i$th coordinate of $Dx \circ y \in \mathbb{R}^d$ is $(Dx)^\top \gamma y$, where $\gamma y$ represents a circular permutation of $y$, in particular the $i$th column of the matrix in equation (3). Now $(Dx)^\top \gamma y = x^\top (\text{diag}(D) \circ \gamma y)$, where $\circ$ represents Hadamard product and $\text{diag}(D) \in \mathbb{R}^d$ is the vector of the diagonal elements of $D$. Denoting $W_i = \text{diag}(D) \circ \gamma y$, the $i$th coordinate of $Dx \circ y$ is obtained by projecting $x$ onto $W_i$. Using Lemma 1, we show that coordinates of $W_i$ are i.i.d. $N(0, 1)$.

Subsequently, we use the same argument presented in the proof of Theorem 1 to demonstrate that the probability that RC:KDTree fails to find exact nearest neighbor of $q$ is identical to that of RPTree.

The preconditioning $Dq \circ y$ via circular convolution operation can be computed in $O(d \log d)$ time with FFT during query processing. The tree traversal time and point processing times are as in Theorem 1, giving us the statement of the theorem. □

**Theorem 3.** Given any $q \in \mathbb{R}^d$, FF:KDTree requires $O(d \log d)$ preconditioning time, $O(\log(n/n_0))$ tree traversal time, $O(dn_0)$ point processing time per tree and ensures that the failure probability of finding the exact nearest neighbor of $q$ is identical to that of RPTree.

**Proof.** Let $H_d G \Pi D$ be as defined in Section 2.2 and let $A \in \mathbb{R}^{d \times d}$ be such that $A = H_d G \Pi D$. Let $A_i \in \mathbb{R}^d$ be the $i$th row of

A. Le et al., 2013 (Section 3.2, [29]) show that coordinates of $A_i$ are i.i.d. $N(0, 1)$. Therefore, for any $x \in \mathbb{R}^d$ (respectively, $q \in \mathbb{R}^d$), the $i$th coordinate of $Ax$ (respectively, $Aq$) can be obtained by projecting $x$ (respectively, $q$) onto random projection direction $A_i$.

With this interpretation and using the argument presented in the proof of Theorem 1, we can say that the probability that FF:KDTree fails to find exact nearest neighbor of $q$ is identical to that of RPTree.

As shown in [29], the preconditioning computation $H_d G \Pi D q$ can be computed in $O(d \log d)$ time – the multiplication with the diagonal matrices $D$ and $G$ can be done in $O(d)$ time; the permutation of a vector of length $d$ (via implicit multiplication with the permutation matrix $\Pi$) takes $O(d)$ time; the multiplication to the Walsh-Hadamard matrix $H_d$ can be done in $O(d \log d)$ time via the Fast Walsh-Hadamard Transform. The tree traversal time and point processing times are as same in Theorem 1. □

**Lemma 1.** Let $u \in \mathbb{R}^d$ be a random vector whose entries are i.i.d. Radamacher variables and let $v \in \mathbb{R}^d$ be a random vector whose entries are i.i.d. $N(0, 1)$. Then $y = u \circ v$ is a random vector in $\mathbb{R}^d$ whose entries are i.i.d. $N(0, 1)$, where $\circ$ represents Hadamard product.

**Proof.** First we show that each coordinate of $y$ follows $N(0, 1)$. Let $y_i$ be the $i$th coordinate of $y$, then $y_i = u_i v_j$. For any scalar $t$:

$$Pr(y_i \leq t) = \mathbb{E}(Pr(v_j \leq t | u_i))$$

$$= Pr(v_j \leq t) Pr(u_i = 1) + Pr(\neg v_j \leq t) Pr(\neg u_i = 1)$$

$$= \frac{1}{2} Pr(v_j \leq t) + \frac{1}{2} Pr(\neg v_j \leq t)$$

$$= Pr(v_j \leq t)$$

where the last equality follows from the fact that $v_j$ and $\neg v_j$ both follow $N(0, 1)$. Therefore, $y_i$ and $v_j$ have the same distribution.

Next, for any $i \neq j$ note that $y_i = u_i v_j$ and $y_j = u_j v_i$. Since $u_i, u_j, v_i, v_j$ are independent of each other, $y_i$ and $y_j$ are independent. Therefore, all coordinates of $y$ are pairwise independent. □

The query time complexities of RR:KDTree and its variants are summarized in Table 2 alongside the guarantees of existing RPTree and variants. We also present the tree construction time complexity and tree space complexity for completeness but skip the formal proof since these can be obtained through standard analyses. The complexities indicate improvements in both time and space for the proposed RC:KDTree and FF:KDTree relative to RPTree.

**4 EMPIRICAL EVALUATIONS**

In this empirical section, we focus on demonstrating that the proposed $kd$-tree based tree index is as effective as the theoretically
Figure 3: Recall vs. Precision curves for RPTree based schemes and the 3 proposed schemes for $k$-nearest-neighbor search with different values of $k$ and leaf size $n_0$ for each tree for all the methods. The curves are generated with $L = 50$ trees.

Implementation details. All the search schemes considered in this empirical evaluation are implemented in the high-level Python language (version 2.7). We use the numpy library for basic linear algebra. For the $F$ and $F^{-1}$ operations needed in RC:KDTree, we use numpy.fft. For the Fast Walsh-Hadamard transform, we use the extremely optimized Fast-Fast Hadamard Transform (FFHT) implementation. All the evaluations were run single-threaded on a 24 core machine with 120GB memory running Ubuntu 18.04.

Datasets. We use various open datasets [7]. The dataset sizes are listed in Table 3 (see appendix Section B). The MNIST, Fashion MNIST, SIFT and GIST datasets are obtained from the ANN benchmark for Euclidean nearest-neighbor search [8]. We use a subset of the 80 million tiny images [42] to create the Tiny Images set.

4.1 Recall vs. Precision

In this comparison, we consider the same values of $(L, n_0)$ for all methods and only perform the defeatist search without sorting the candidates we get from defeatist search. So, for $l$ trees, the search time of the defeatist search with RPTree would be $O(l d \log n)$ instead of $O(l d \log n + l n_0)$. The reason for this is because the

---

(a) $k = 10$, $n_0 = 10$

(b) $k = 10$, $n_0 = 20$

(c) $k = 100$, $n_0 = 100$

(d) $k = 100$, $n_0 = 200$
constant term $dn_0$ is same for all the methods we are considering in our empirical evaluation. Then we compute the precision and recall for each $l = 1, \ldots, L$ to generate the recall vs. precision curves for all methods. Figure 3 presents the results on the 8 datasets for $k$-nearest-neighbor search, with trees of leaf size at most $n_0$ and $L = 50$ for different values of $k$, $n_0$. These results indicate that there is no apparent difference in the search accuracy-efficiency trade-offs between the proposed schemes and the baselines, the theoretical guarantees for RR:KDTree, RC:KDTree and FF:KDTree are validated in practice. The area under the curves for $k = 100$, $n_0 = 100$ are presented in the appendix C in Table 4 to demonstrate how similar the recall vs. precision trade off performances are.

4.2 Tree traversal time scaling with $n$ and $d$

Combining the SIFT, GIST and Tiny Images data sets, we create data sets with $n = 1024, 2048, 4096, \ldots, 1000000$ and $d = 2^l, l = 7, \ldots, 11$ to evaluate the scalability of the proposed schemes and the baseline(s) (see dataset details in appendix Section B). We used a subset of size 25000 as the set of queries. The purpose of this evaluation is to show the combined scaling of the search algorithms (or rather implementation of the algorithms) with changing $d$ and $n$. For this purpose, we consider the query time of all algorithms at $d = 128$, $n = 1024$ as their respective baselines and study the relative increase in run time with increasing $d$ as well as $n$. For this evaluation, we only focus on the RPTree baseline (with $O(d \log n)$ scaling) and the proposed RC:KDTree and FF:KDTree (with $O(d \log d + \log n)$ scaling). The scaling is presented in Figure 4. The values reported in the results are the ratios of the actual query run time with that of the method at $n = 1024, d = 128$ with increasing $n$ (horizontal axis) and increasing $d$ (different markers). This allows us to see the effect of increasing $n$ and $d$ collectively.

The results demonstrate the favorable scaling of the proposed RC:KDTree and FF:KDTree with respect to $n$ for a fixed $d$ relative to that of RPTree. This is because of the fact, that for RC:KDTree and FF:KDTree, the $O(d \log d)$ pre-conditioning is independent of $n$, and the subsequent tree traversal cost for the $kd$-tree scales only at $O(\log n)$ independent of $d$. In contrast, RPTree query time increases (relatively) more significantly because of the $O(d \log n)$ scaling where the cost of increasing $n$ gets multiplied by $d$.

The scaling with respect to $d$ for the proposed RC:KDTree and FF:KDTree is not favorable relative to RPTree. However, one of the major reasons for this is that the actual runtimes are very dependent on the implementations, and we made no attempt to optimize our implementation beyond the available reference tools. The vector-vector dot-product operation, which is the main computational step is RPTree is extremely optimized in the standard linear algebra packages like numpy – the vector-vector or the vector-matrix multiplication is a super cache friendly operation because it is possible to stream all the operations straight through all the required memory without any cache misses, and potentially few waits on cold data. This is demonstrated by the $d$-scaling of the implementation of the $O(d \log n)$ RPTree for a fixed $n$ – increasing $d 16\times$ (from 128 to 2048) only increases the actual run time by less than 2x even though the run time complexity of the RPTree defeatist search is pretty tight. In contrast, the FFT reference implementation isumpy is not optimized. For example, the sin/cos tables involved in the

4.3 Index size scaling with $n$ and $d$

The tree index size scaling is presented in Figure 6 in appendix Section E. The results indicate that for smaller values of $n$, RC:KDTree and FF:KDTree have much smaller tree index size in memory compared to RPTree. But as $n$ grows, the index size is dominated by the terms linear in $n$, making the index sizes of all three schemes very close to each other. Note that the proposed schemes RC:KDTree and FF:KDTree do not ever have larger index size than RPTree.

5 CONCLUDING REMARKS

We proposed a way to make $kd$-tree based nearest-neighbor search theoretically (and empirically) competitive in high dimensions, resulting in a scheme with $O(d \log d + \log n)$ query time. This scheme can be used for improved theoretical and empirical scaling of RPTree for the problem of maximum inner-product search [27, 28]. However, this is still some ways off from the $O(\log n)$ query time of $kd$-tree in FLANN [36] not withstanding the lack of any theoretical guarantees for FLANN. We believe that our proposed scheme can be improved upon to get a $kd$-tree based search scheme with close to $O(\log n)$ scaling while still retaining the search accuracy guarantees. In addition to improved theoretical guarantees, we also plan to create and open-source an optimized implementation of the proposed schemes that is implemented in a low-level language (such as C/C++) and makes use of most efficient available libraries (such as FFTW for FFT and inverse FFT). Moreover, we plan to explore (i) auxiliary information and priority functions [26] to improve the overall space complexity, and (ii) inverted multi-index [9] with $kd$-tree for improved search efficiency.
REFERENCES


A  CHOICE OF LEAF SIZE \(n_0\)

The leaf size \(n_0\), in some sense, balances between a global search and a local search. The whole idea is to quickly identify a local region (which most likely will contain nearest neighbors) and perform exhaustive search there, as opposed to performing a linear scan (global search). Of course making \(n_0\) too small will make the local region uninformative (may be too small and may not contain true nearest neighbors) while making \(n_0\) too big would result in unnecessary exhaustive search within this local region increasing query time. While this problem sounds kind of similar to the same issue that LSH faces (in terms of requiring knowledge of true nearest-neighbor distance or else ending up with disparate candidate set sizes for different queries), typically effect of \(n_0\) on search accuracy is quite benign. Assuming data is generated from an underlying probability distribution which is a “doubling measure” with doubling constant \(d_0\) (that is, probability mass of any ball can be covered by \(2^{d_0}\) balls of half the radius, where \(d_0\) is much smaller than ambient data dimension \(d\)), a constant failure probability of finding all true \(k\) nearest neighbors requires \(n_0 \sim O\left(\left(d_0 k d_0\right)^{d_0}\right)\) (see Theorem 9 in [14] for exact dependence).

B  DATASET DETAILS

The sizes of the datasets used for the empirical evaluation are presented in Table 3.

| Data set        | \(|R|\)   | \(|Q|\)   | \(d\) |
|-----------------|---------|---------|-------|
| Letter recognition | 18000   | 2000    | 16    |
| Corel           | 56615   | 10000   | 89    |
| Aerial          | 265465  | 10000   | 60    |
| Tiny images     | 1000000 | 10000   | 384   |
| MNIST           | 60000   | 10000   | 784   |
| Fashion MNIST   | 60000   | 10000   | 784   |
| SIFT            | 1000000 | 10000   | 128   |
| GIST            | 1000000 | 1000    | 960   |

**Dataset details for the scaling experiment.** For \(d = 128\), we just used SIFT. For \(d = 256\), we randomly sampled \(256/384\) columns in the Tiny Images set. For \(d = 512\), we column-subsampled the GIST dataset. For \(d = 1024\), we combined SIFT and GIST and then subsampled \(1024\) out of the \(1088\) columns. For \(d = 2048\), we used \(3\) million rows from the \(80\) million tiny images set, and concatenated them to get \(1\) million rows with \(1152\) columns, and then concatenated that to the \(1024\) column set of GIST + SIFT and subsampled \(2048\) out of the \(2176\) columns. The query set was created in a similar consistent manner to ensure that there was no overlap in the data between the query and reference sets.

C  FURTHER RESULTS ON RECALL VS. PRECISION

The area under the recall-precision curves in Figure 3 is presented in Table 4.

D  IMPLEMENTATION DETAILS

Here we present the actual runtimes of the different preconditioning operations we have considered in the different methods. Table 5 details the operations we will be focusing on. The goal of this section is to show that the actual run time of the implementations can be significantly different than their theoretical runtimes because of optimizations in the implementation. The results in Figure 5 indicate that the \(O(d^2)\) MatMul operation (used by RR:KDTree) is the fastest for up to \(d \approx 2^9\) (the \(O(d \log(n/n_0))\) query time of RPTree will only be faster than \(O(d^2)\)) and only passes the \(O(d \log d)\) CC_x operation (used in RC:KDTree) for \(d > 2^9\). As mentioned in Section 4.2, the vector-vector/vector-matrix multiplication can be really fast in practice. For example, increasing the dimension (horizontal axis) by \(4\times\) \((2^4 - 2^6)\) increases the run time (vertical axis) by less than \(2\times\) (instead of increasing run time by \(16\times\)). This is one of the reasons for the (relatively) favorable \(d\) scaling of RPTree in Section 4.2. The \(O(d^2)\) scaling of MatMul starts showing up for \(d > 2^9\).

In contrast, our un-optimized implementation of our proposed schemes (CC_x for RC:KDTree and FF_x for FF:KDTree) suffers from various inefficiencies:

- As mentioned in Section 4.2, we use a reference implementation of FFT instead of the optimized FFTW library\(^8\).
- We use the standard numpy random permutation (with fixed seed) for the permutation operation in FF_x. This can be significantly improved with a custom C++ implementation.
- Moreover, our operations (CC_x and FF_x in Table 5) switches between low-level (C/C++) and high level (Python) significantly more than the standard vector-vector/vector-matrix operations, which comes with significant overheads as well:
  - Something like \(\Gamma x\) (in RR:KDTree) and \(w^\top x\) (in RPTree) can be obtained by a single round trip between the high-level Python layer and the low-level numpy implementations.

\(^{8}\text{http://fftw.org}\)
Method | Aerial | Corel | FashMNIST | GIST | LetterReg | MNIST | SIFT | TinyImages
---|---|---|---|---|---|---|---|---
RPTree | 0.040 ± 0.001 | 0.076 ± 0.004 | 0.038 ± 0.001 | 5.9e-4 ± 3e-5 | 0.139 ± 0.003 | 0.030 ± 0.000 | 6.53e-3 ± 12e-5 | 1.45e-3 ± 3e-5
SpGa:RPT(1/10) | 0.040 ± 0.001 | 0.077 ± 0.001 | 0.039 ± 0.001 | 6.0e-4 ± 2e-5 | 0.135 ± 0.011 | 0.031 ± 0.001 | 6.35e-3 ± 11e-5 | 1.41e-3 ± 3e-5
SpGa:RPT(1/5) | 0.041 ± 0.002 | 0.074 ± 0.002 | 0.039 ± 0.001 | 6.0e-4 ± 3e-5 | 0.141 ± 0.005 | 0.030 ± 0.000 | 6.50e-3 ± 8e-5 | 1.46e-3 ± 4e-5
SpRa:RPT(1/10) | 0.041 ± 0.000 | 0.074 ± 0.001 | 0.039 ± 0.001 | 5.9e-4 ± 1e-5 | 0.136 ± 0.007 | 0.030 ± 0.001 | 6.35e-3 ± 9e-5 | 1.43e-3 ± 5e-5
SpRa:RPT(1/3) | 0.042 ± 0.001 | 0.075 ± 0.003 | 0.039 ± 0.001 | 6.0e-4 ± 2e-5 | 0.139 ± 0.012 | 0.030 ± 0.001 | 6.25e-3 ± 10e-5 | 1.46e-3 ± 4e-5
RR:KDTree | 0.041 ± 0.001 | 0.074 ± 0.002 | 0.039 ± 0.001 | 6.1e-4 ± 1e-5 | 0.144 ± 0.005 | 0.030 ± 0.001 | 6.54e-3 ± 8e-5 | 1.43e-3 ± 1e-5
RC:KDTree | 0.042 ± 0.002 | 0.073 ± 0.003 | 0.039 ± 0.001 | 6.0e-4 ± 2e-5 | 0.137 ± 0.003 | 0.031 ± 0.001 | 6.51e-3 ± 10e-5 | 1.40e-3 ± 4e-5
FF:KDTree | 0.039 ± 0.001 | 0.076 ± 0.003 | 0.037 ± 0.001 | 5.4e-4 ± 4e-5 | 0.134 ± 0.007 | 0.030 ± 0.000 | 6.36e-3 ± 15e-5 | 1.44e-3 ± 3e-5

Table 4: Area under the recall vs. precision curves presented in Figure 3 averaged over 5 runs for \(k\)-nearest-neighbor search with \(k = 100\) and the leaf size \(n_0 = 100\) for each tree for all the methods.

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Complexity</th>
<th>Implementation (in Python)</th>
<th>High-low round trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD_x</td>
<td>(H_d X)</td>
<td>(O(d \log d))</td>
<td>ffht.fht(D * x)</td>
<td>2</td>
</tr>
<tr>
<td>MatMul</td>
<td>(\Gamma x)</td>
<td>(O(d^2))</td>
<td>np.dot((\Gamma), (x))</td>
<td>1</td>
</tr>
<tr>
<td>CC_x</td>
<td>(D_x \odot \gamma)</td>
<td>(O(d \log d))</td>
<td>np.fft.ifft(np.fft.fft(D * x) * np.fft.fft((\gamma)))</td>
<td>4</td>
</tr>
<tr>
<td>FF_x</td>
<td>(H_d G_1 H_d X)</td>
<td>(O(d \log d))</td>
<td>ffht.fht(G * np.random.permutation(ffht.fht(D * x)))</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5: The details of the different preconditioning operations we have considered in Table 1 and implemented for the empirical evaluation. Here \(x \in \mathbb{R}^d\) with \(d = 2^l\) for some integer \(l > 0\). \(\text{np}\) stands for the \text{numpy} linear algebra package in Python. \text{ffht} is the FFHT package in the FALCONN-LIB library. The actual computation times of the implementations considered are presented in Figure 5. Note that we cache the value of \(\text{np.fft.fft}(\(\gamma\))\) instead of computing it for every circular convolution.

To create a level playing field between all these operations, we believe that all the preconditioning operations need to be completely implemented in a low-level language (such as C/C++) and the preconditioning operation can be computed in a single high-low level round trip as in the case of the \(O(d^2)\) matrix multiplication or the \(O(d)\) dot product.

### E. TREE INDEX SIZE SCALING

The tree index size scaling is presented in Figure 6.

---

9https://github.com/FALCONN-LIB/FFHT