

Revisiting *kd*-trees for Nearest-Neighbor Search

Parikshit Ram (IBM Research AI)

Kaushik Sinha (Wichita State University)



What is nearest-neighbor search?

→ A set of points $S \subset \mathbb{R}^d$, $|S| = n$

→ A search query $q \in \mathbb{R}^d$

→ Measure of similarity $d(q, p)$

→ Nearest neighbor search

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

→ k -nearest neighbor search

$$P_k^* = \arg \min_{P \subset S, |P|=k} \max_{p \in P} d(q, p)$$

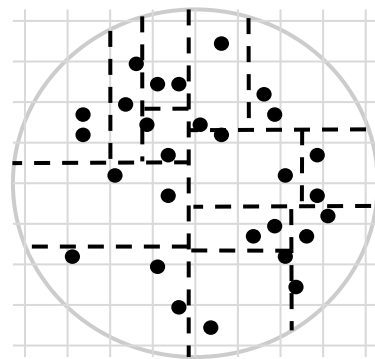
→ Euclidean nearest-neighbor search $d(q, p) = \|q - p\|_2$

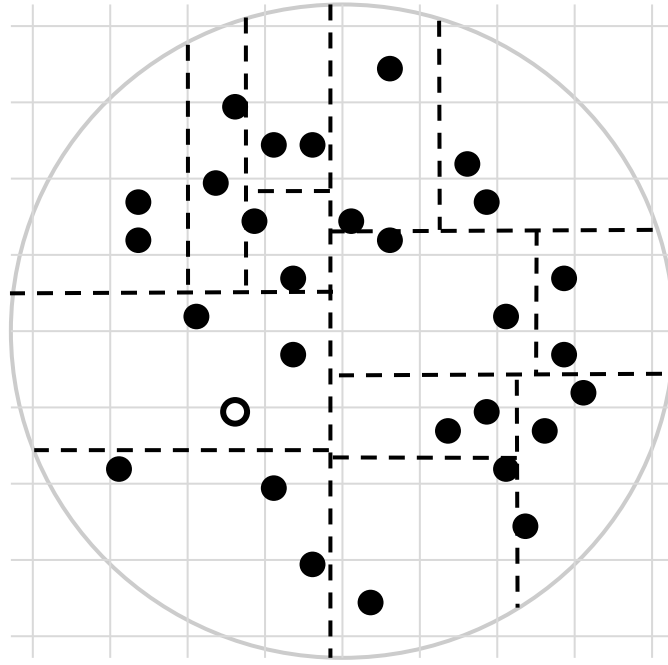
Search techniques

- Space partitioning trees -- *kd*-trees, metric trees, cover trees, PCA trees, ...
 - ◆ Exact; unfavorable theoretical guarantees; empirical performance can be good
- Locality preserving hashing schemes -- Locality sensitive hashing (LSH)
 - ◆ Approximate; favorable theoretical guarantees; empirical performance unpredictable; hard to control precision-recall tradeoff
- Data dependent quantization -- Product quantization
 - ◆ Approximate; no theoretical guarantees; strong empirical performance; good control over precision-recall tradeoff
- Similarity graphs -- Hierarchical graph traversal
 - ◆ Approximate; no theoretical guarantees; strong empirical performance; good control over precision-recall tradeoff

Search with *kd*-tree

- Axis aligned space partition
- Search with *backtracking depth-first tree traversal*
- Pros
 - ◆ Exact search
 - ◆ Logarithmic dependence on number of points
 - ◆ Fast in low dimensions
 - ◆ Low memory overhead
- Cons
 - ◆ Exponential dependence in data dimensionality
 - ◆ No advantage over brute force in moderate to high dimensions

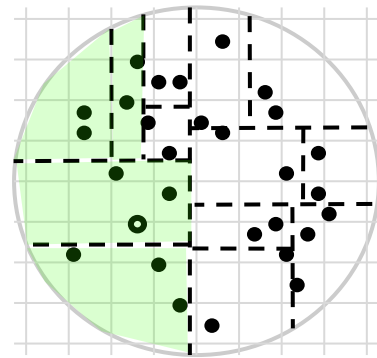
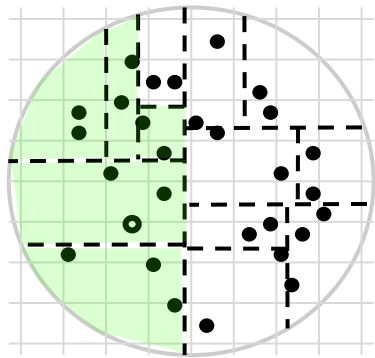
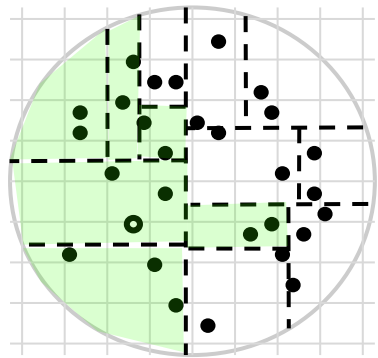
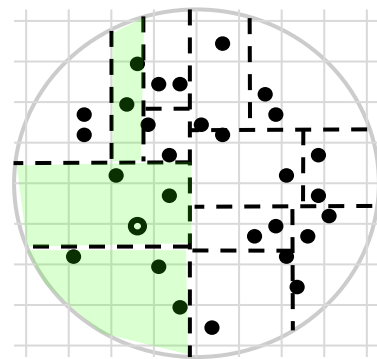
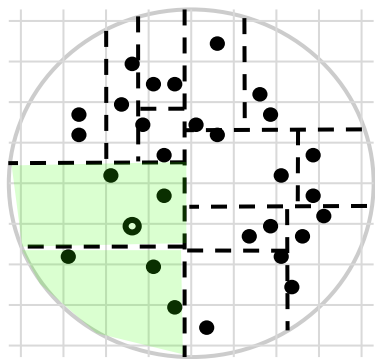
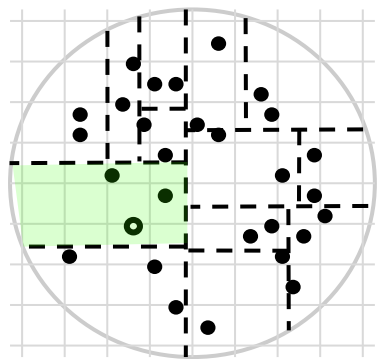




→ Index size $O(n)$

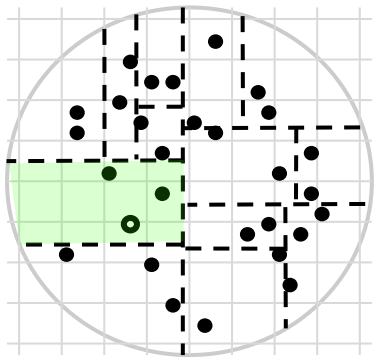
→ Search time complexity $O(\log n)$

(for small number of dimensions)

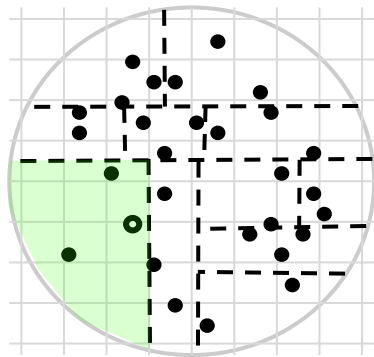


Search with ensemble of *kd*-trees (FLANN)

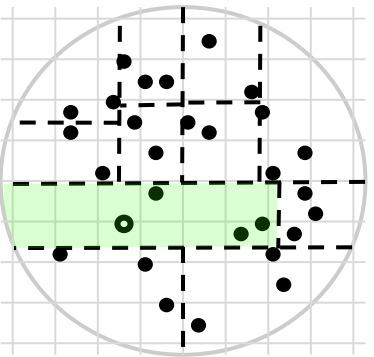
- Space partitioned with randomly generated axis-aligned splits
- An ensemble of trees built instead of a single one
- Search with *defeatist tree traversal* across all trees
- Pros
 - ◆ Very fast in practice, competitive even in high dimensions
 - ◆ Still low memory overhead
- Cons
 - ◆ Approximate, with no theoretical guarantees on search accuracy



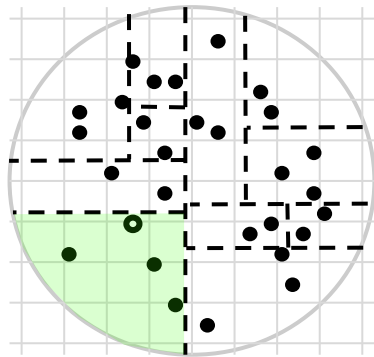
Tree 1



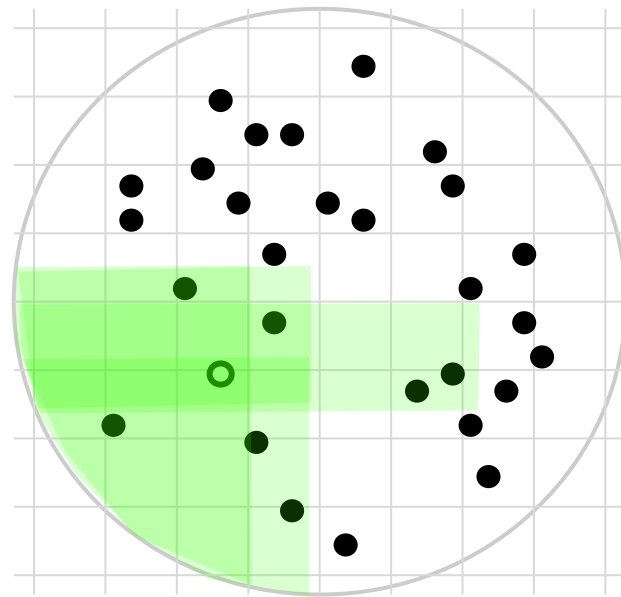
Tree 2



Tree 3



Tree 4

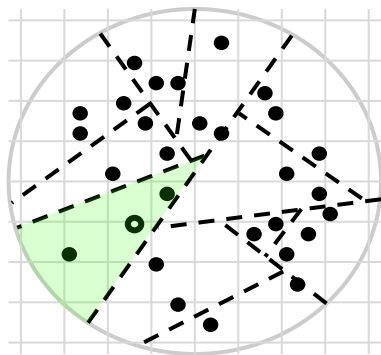


Candidates

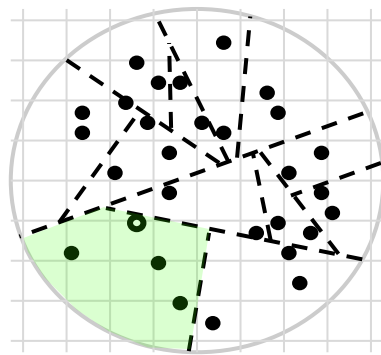
- Index size $T \times O(n)$
- Search time $T \times O(\log n)$

Search with Randomized Partition Trees (RPTree)

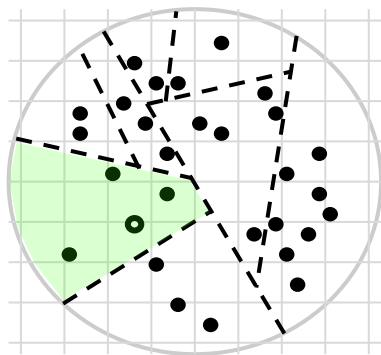
- Space partitioned with random directions
- An ensemble of trees built instead of a single one
- Search with defeatist tree traversal across all trees
- Pros
 - ◆ Fairly competitive (outperforms LSH) in practice
 - ◆ Rigorous theoretical guarantees on search accuracy
- Cons
 - ◆ Relatively high memory overhead
 - ◆ Theoretical runtime not as fast as FLANN



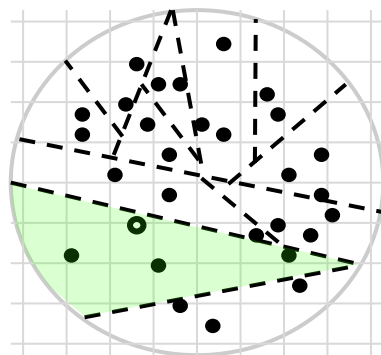
Tree 1



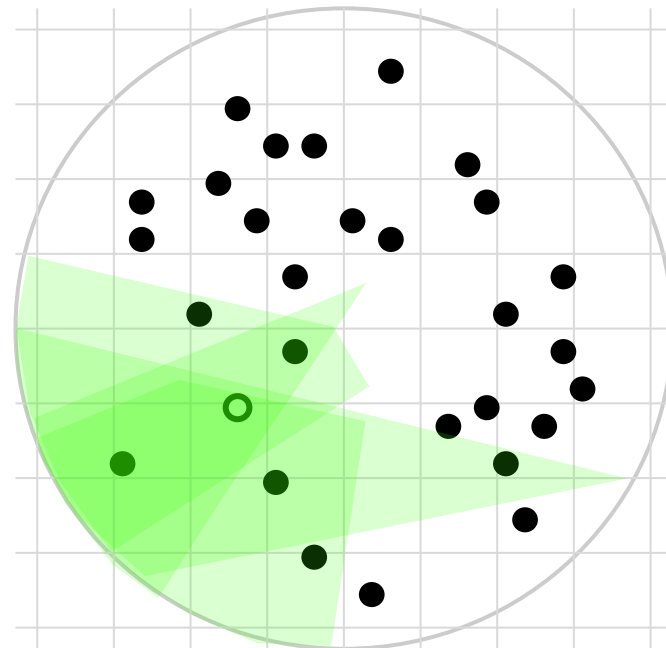
Tree 2



Tree 3



Tree 4



Candidates

- Index size $T \times O(d \log n + n)$
- Search time $T \times O(d \log n)$

Our main contributions

→ Improved search runtime **with *kd*-trees**

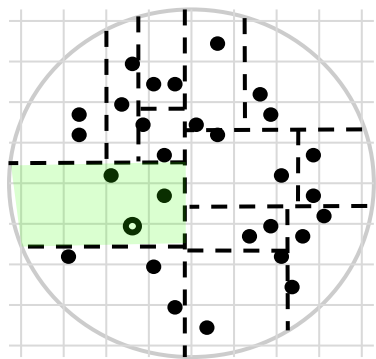
$$O(d \log n) \longrightarrow O(d \log d + \log n)$$

→ Improved index size **with *kd*-trees**

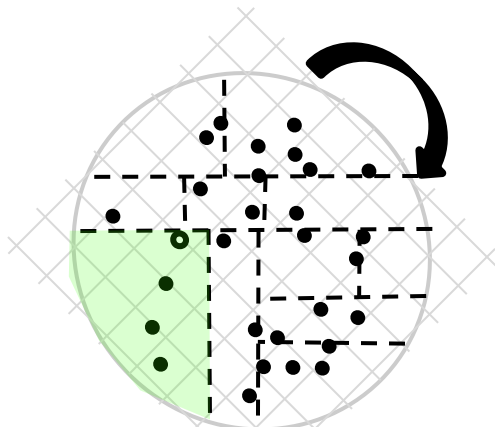
$$O(d \log n + n) \longrightarrow O(d + n)$$

Randomized rotation + *kd*-tree (RR:*kd*-tree)

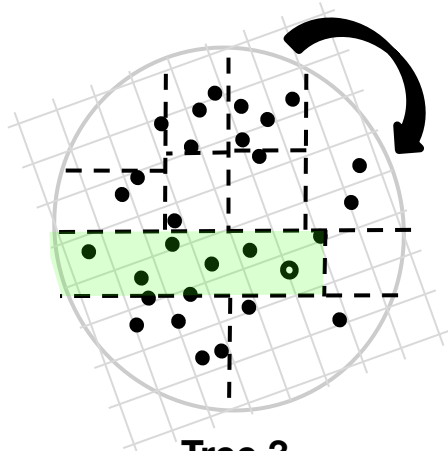
- Random rotate the data
- Space partitioned with *kd*-trees (axis-aligned splits)
- An ensemble of pairs of (random rotation + *kd*-tree)
- Search by (random rotation + defeatist tree traversal) across all pairs
- Pros
 - ◆ Theoretical search accuracy guarantees **equivalent to RPTree** (Theorem 1)
- Cons
 - ◆ Quadratic dependence on dimension on search runtime and memory overhead



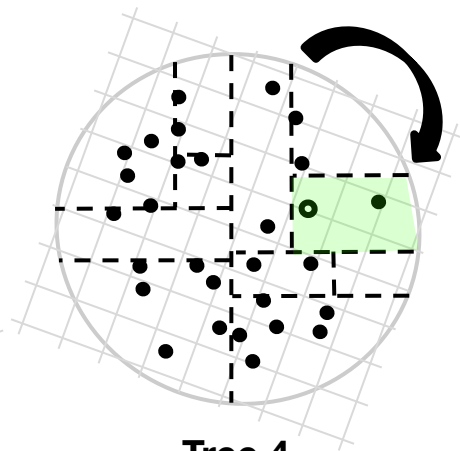
Tree 1



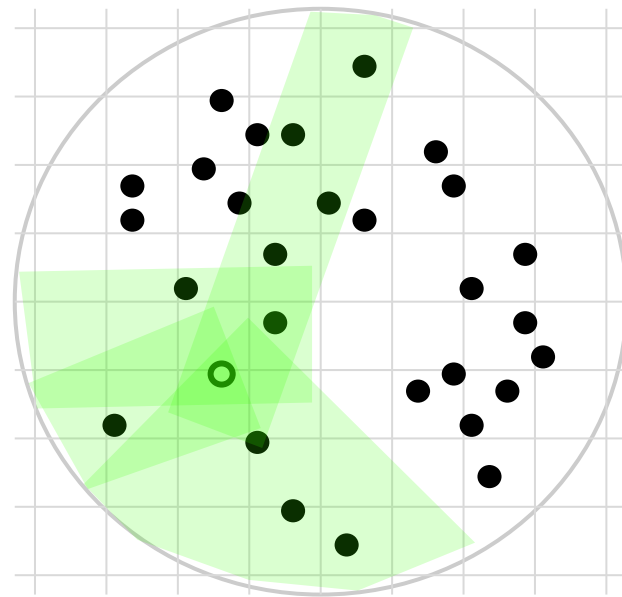
Tree 2



Tree 3



Tree 4



Candidates

- Index size $T \times O(d^2 + n)$
- Search time $T \times O(d^2 + \log n)$

RPTree vs. RR: kd -tree

→ Search runtime per tree

$$O(d \log n) \longrightarrow O(d^2 + \log n)$$

→ Index size per tree

$$O(d \log n + n) \longrightarrow O(d^2 + n)$$

Randomized rotation too expensive!

Randomized rotation

For any point $\mathbf{p} \in \mathbb{R}^d$

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \dots & \gamma_{1d} \\ \gamma_{21} & \gamma_{22} & \dots & \gamma_{2d} \\ \dots & \dots & \dots & \dots \\ \gamma_{d1} & \gamma_{d2} & \dots & \gamma_{dd} \end{bmatrix} \mathbf{p}, \gamma_{ij} \sim \mathcal{N}(0, 1)$$

Random rotation matrix

Approximating randomized rotation

Randomized circular convolution

$$\underbrace{\begin{bmatrix} \gamma_1 & \gamma_d & \dots & \gamma_2 \\ \gamma_2 & \gamma_1 & \dots & \gamma_3 \\ \dots & \dots & \dots & \dots \\ \gamma_{d-1} & \gamma_{d-2} & \dots & \gamma_d \\ \gamma_d & \gamma_{d-1} & \dots & \gamma_1 \end{bmatrix}}_{\text{Random circulant matrix}} \underbrace{\begin{bmatrix} -1 & 0 & \dots & 0 & 0 \\ 0 & +1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & +1 & 0 \\ 0 & 0 & \dots & 0 & -1 \end{bmatrix}}_{\text{Random diagonal sign matrix}} \quad p, \gamma_i \sim \mathcal{N}(0, 1)$$

Approximating randomized rotation

$$\begin{bmatrix} \gamma_1 & \gamma_d & \dots & \gamma_2 \\ \gamma_2 & \gamma_1 & \dots & \gamma_3 \\ \dots & \dots & \dots & \dots \\ \gamma_{d-1} & \gamma_{d-2} & \dots & \gamma_d \\ \gamma_d & \gamma_{d-1} & \dots & \gamma_1 \end{bmatrix} \begin{bmatrix} -1 & 0 & \dots & 0 & 0 \\ 0 & +1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & +1 & 0 \\ 0 & 0 & \dots & 0 & -1 \end{bmatrix} \quad p, \gamma_i \sim \mathcal{N}(0, 1)$$

can be written as

$$\gamma \circledast (D \circ p)$$

→ Element-wise product
→ Circular convolution

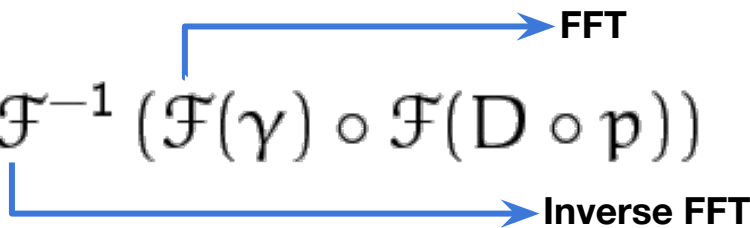
where

$$\gamma = [\gamma_1, \dots, \gamma_d]^\top, \gamma_i \sim \mathcal{N}(0, 1),$$

$$D = [D_1, \dots, D_d]^\top, D_i \sim \mathcal{R} \rightarrow \text{Radamacher random variable}$$

Fast approximate randomized rotation

Fast convolution via Fast Fourier Transform

$$\gamma \circledast (\mathbf{D} \circ \mathbf{p}) = \mathcal{F}^{-1} (\mathcal{F}(\gamma) \circ \mathcal{F}(\mathbf{D} \circ \mathbf{p}))$$


The diagram shows two blue arrows indicating the steps in the equation. One arrow starts from the top of the \mathcal{F}^{-1} term and points to the right, labeled "FFT". The other arrow starts from the bottom of the \mathcal{F}^{-1} term and points to the right, labeled "Inverse FFT".

$$\gamma = [\gamma_1, \dots, \gamma_d]^\top, \gamma_i \sim \mathcal{N}(0, 1),$$

$$\mathbf{D} = [\mathbf{D}_1, \dots, \mathbf{D}_d]^\top, \mathbf{D}_i \sim \mathcal{R}$$

Fast approximate randomized rotation

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1d} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2d} \\ \cdots & \cdots & \cdots & \cdots \\ \gamma_{d1} & \gamma_{d2} & \cdots & \gamma_{dd} \end{bmatrix} \mathbf{p} \approx \mathcal{F}^{-1} (\mathcal{F}(\gamma) \circ \mathcal{F}(D \circ \mathbf{p}))$$

$$O(d^2) \longrightarrow O(d \log d)$$

RPTree vs. RC:*kd*-tree - Improved runtime

→ Improved search runtime per tree

$$O(d \log n) \longrightarrow O(d \log d + \log n)$$

→ Improved index size per tree

$$O(d \log n + n) \longrightarrow O(d + n)$$

→ Theoretical search accuracy guarantee

equivalent to RR:*kd*-tree, hence equivalent to RPTree

(Theorem 2)

Randomized rotation

For any point $\mathbf{p} \in \mathbb{R}^d$

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \dots & \gamma_{1d} \\ \gamma_{21} & \gamma_{22} & \dots & \gamma_{2d} \\ \dots & \dots & \dots & \dots \\ \gamma_{d1} & \gamma_{d2} & \dots & \gamma_{dd} \end{bmatrix} \mathbf{p}, \gamma_{ij} \sim \mathcal{N}(0, 1)$$

Random rotation matrix

Approximating randomized rotation II

Approximately Gaussian random matrix with FastFood

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \dots & \gamma_{1d} \\ \gamma_{21} & \gamma_{22} & \dots & \gamma_{2d} \\ \dots & \dots & \dots & \dots \\ \gamma_{d1} & \gamma_{d2} & \dots & \gamma_{dd} \end{bmatrix} \approx H_d G \Pi H_d D$$

Approximating randomized rotation II

Approximately Gaussian random matrix with FastFood

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1d} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2d} \\ \cdots & \cdots & \cdots & \cdots \\ \gamma_{d1} & \gamma_{d2} & \cdots & \gamma_{dd} \end{bmatrix}$$

$$\approx H_d G \Pi H_d D$$

Walsh-Hadamard matrix

Random permutation matrix

Random diagonal sign matrix

Random diagonal Gaussian matrix

Approximating randomized rotation II

Approximately Gaussian random matrix with FastFood

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1d} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2d} \\ \cdots & \cdots & \cdots & \cdots \\ \gamma_{d1} & \gamma_{d2} & \cdots & \gamma_{dd} \end{bmatrix} \approx H_d G \Pi H_d D$$

$$G = \text{diag}([G_1, \dots, G_d]), G_i \sim \mathcal{N}(0, 1)$$

$$D = \text{diag}([D_1, \dots, D_d]), D_i \sim \mathcal{R}$$

$$\Pi = \text{random permutation matrix}$$

$$H_d = \text{Walsh-Hadamard matrix}$$

$$H_{2d} := \begin{bmatrix} H_d & H_d \\ -H_d & H_d \end{bmatrix}, H_1 := 1$$

Fast approximate randomized rotation II

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1d} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2d} \\ \cdots & \cdots & \cdots & \cdots \\ \gamma_{d1} & \gamma_{d2} & \cdots & \gamma_{dd} \end{bmatrix} p \approx (H_d G \Pi H_d D) p$$

$$O(d^2) \longrightarrow O(d \log d)$$

$Dp \rightarrow O(d)$
$H_d(Dp) \rightarrow O(d \log d)$
$\Pi(H_d Dp) \rightarrow O(d)$
$G(\Pi H_d Dp) \rightarrow O(d)$
$H_d(G \Pi H_d Dp) \rightarrow O(d \log d)$

RPTree vs. FF:*kd*-tree - Improved runtime II

→ Improved search runtime

$$O(d \log n) \longrightarrow O(d \log d + \log n)$$

→ Improved index size

$$O(d \log n + n) \longrightarrow O(d + n)$$

→ Theoretical search accuracy guarantee

equivalent to RR:*kd*-tree, hence equivalent to RPTree

(Theorem 3)

Search accuracy performance

→ Precision recall curves

→ Baselines

- ◆ RPTree
- ◆ Sparse RPTree (4 versions)

→ Proposed

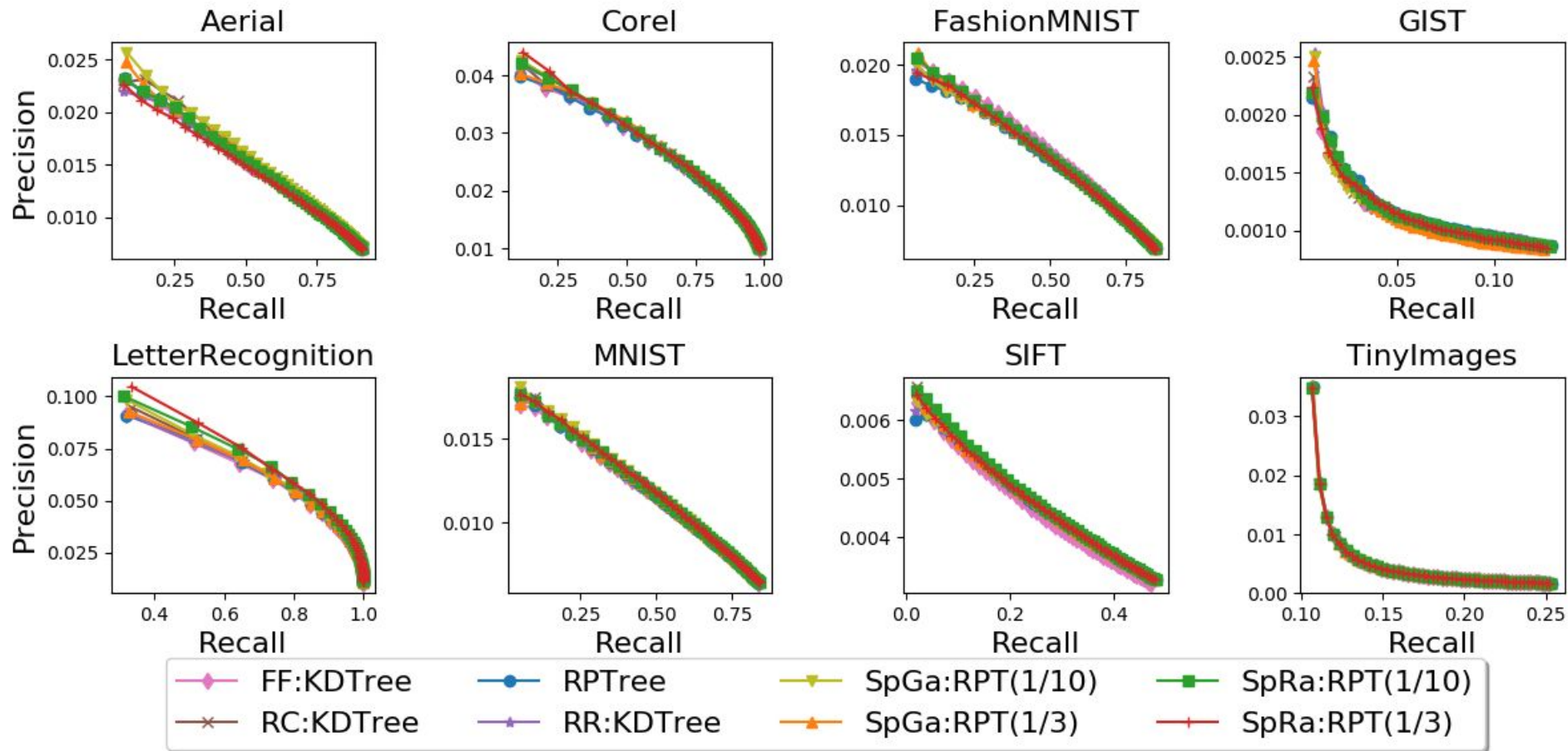
- ◆ RR:*kd*-tree
- ◆ RC:*kd*-tree
- ◆ FF:*kd*-tree

→ Code:

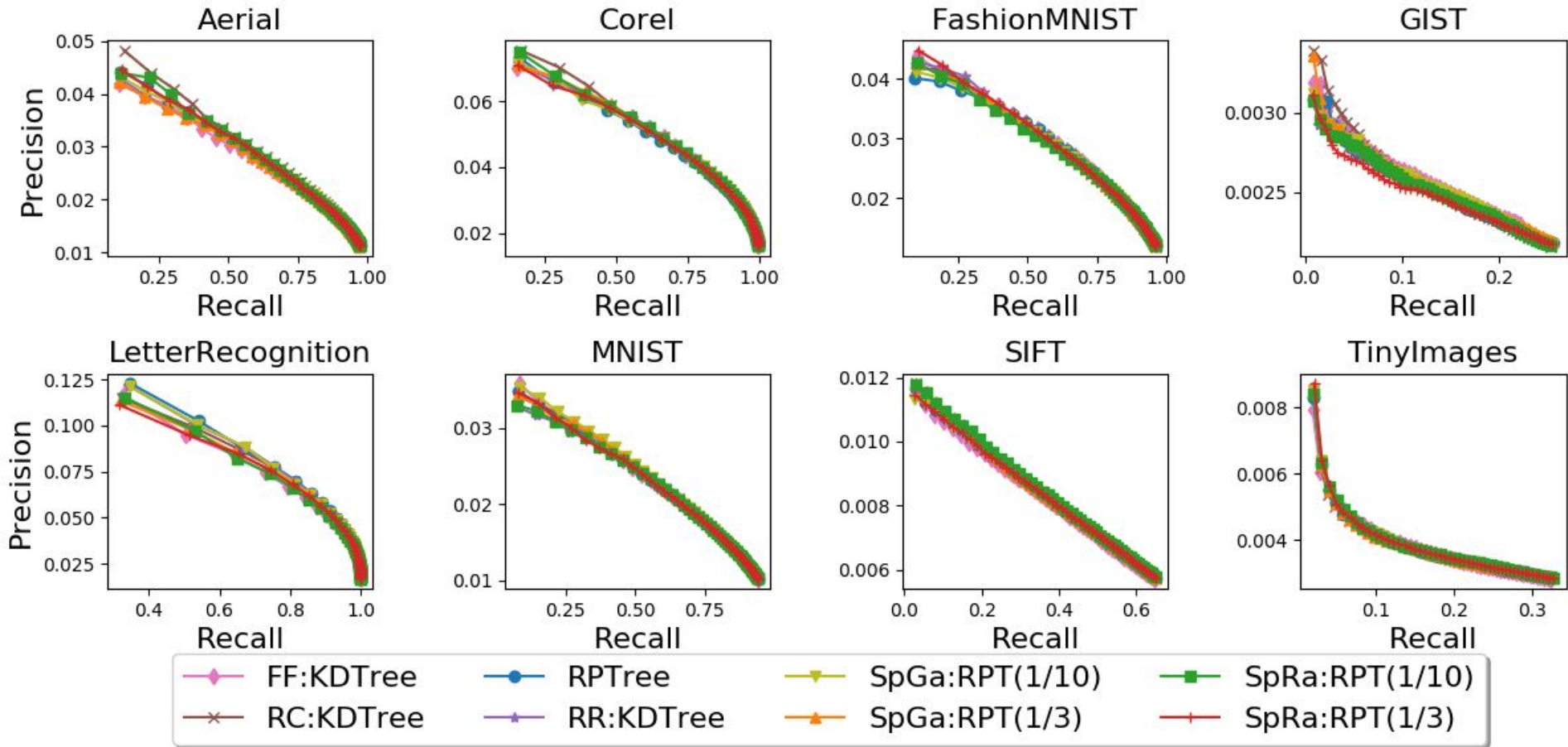
<https://github.com/rithram/rrkdt>

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

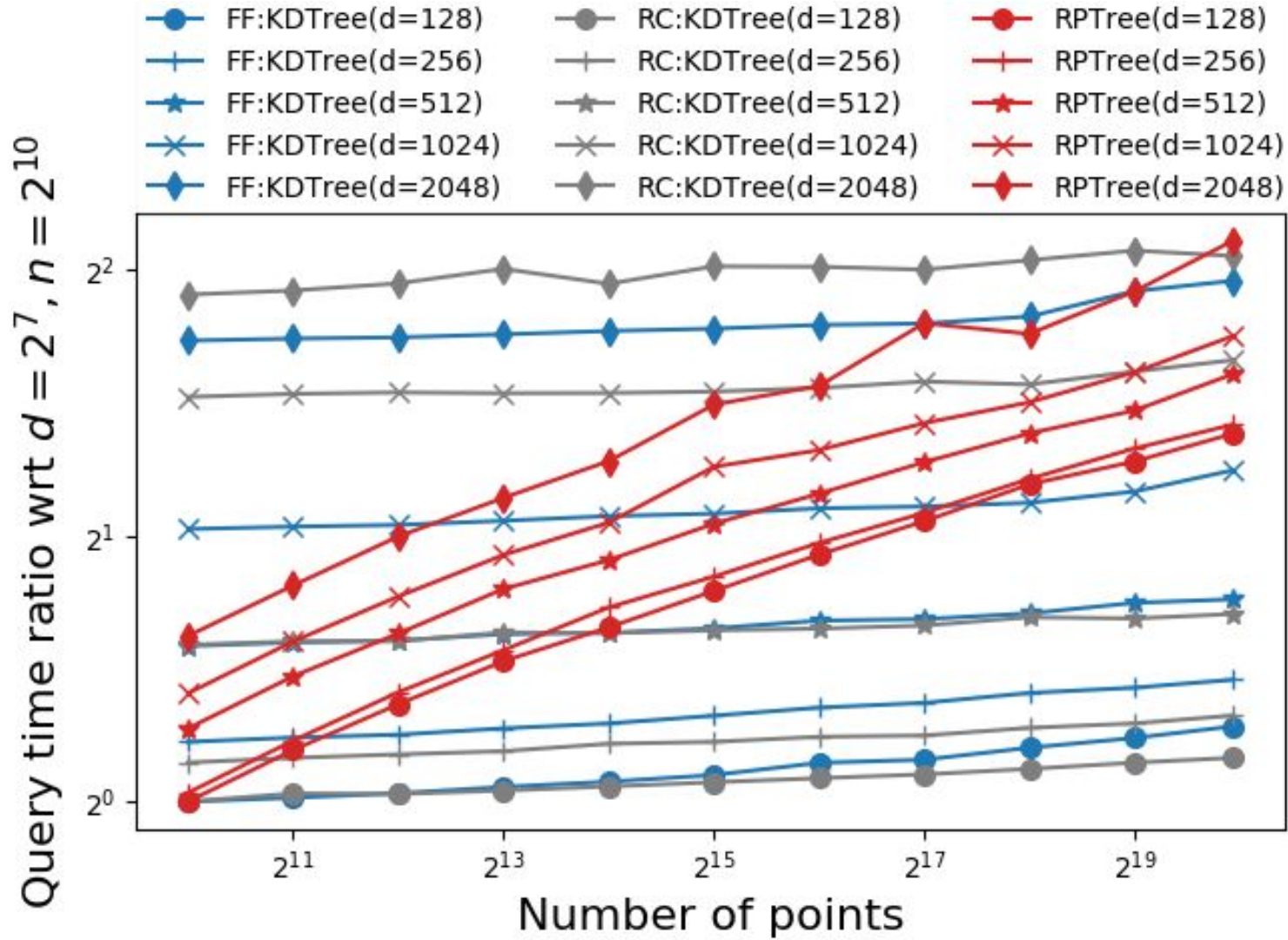
10 neighbors, maximum leaf size 20



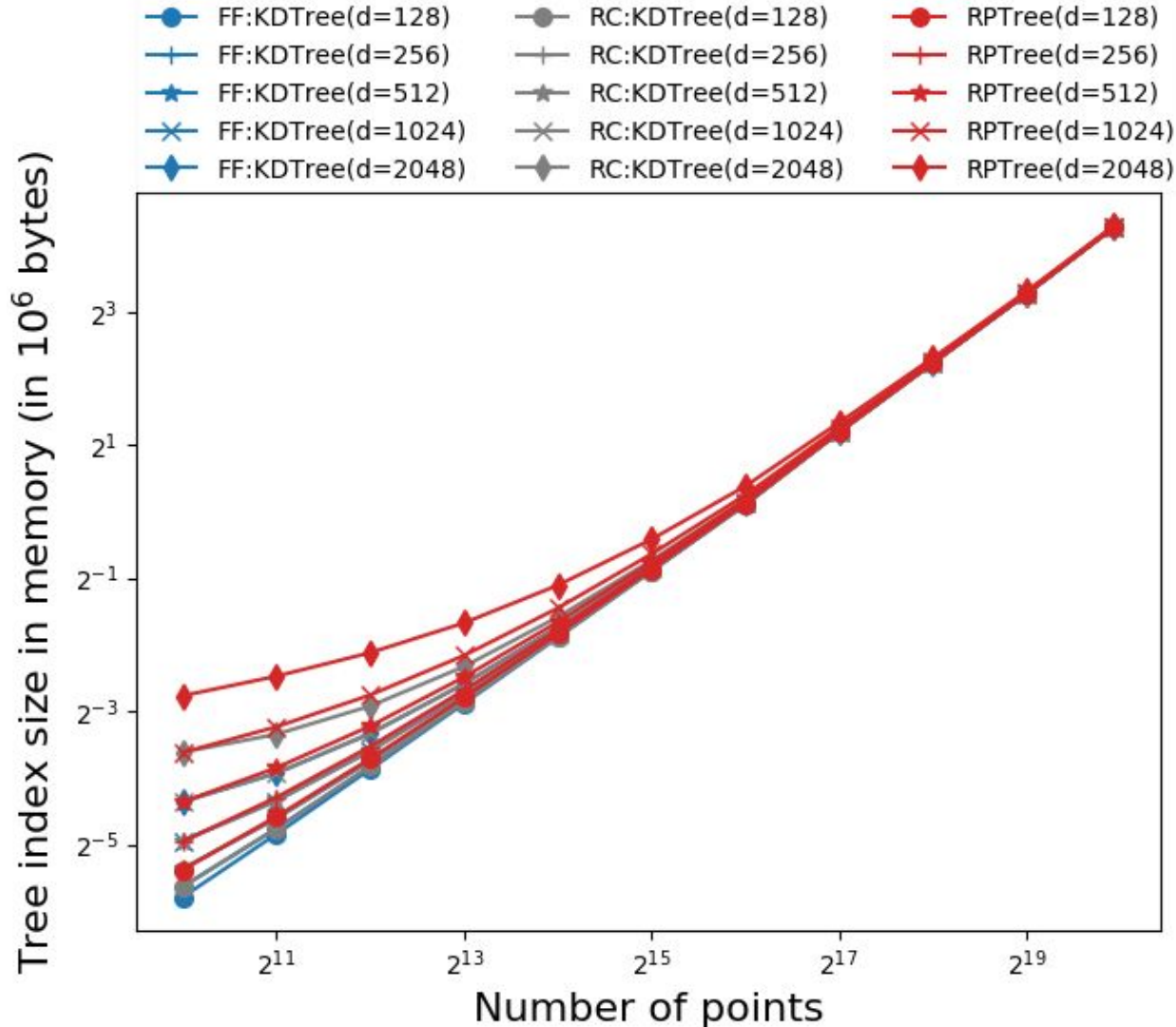
100 neighbors, maximum leaf size 200



Search time scaling



Index size scaling



Future directions

- Improve runtime and memory scaling even further
(while retaining guarantees)

$$O(Td \log d + T \log n) \longrightarrow O(d \log d + T \log n)$$

- High performance C/C++ implementation with state-of-the-art FFT/FWHT
- Explore inverted multi-index for improved search

Thank you!

→ Contact

◆ p.ram@gatech.edu

◆ Kaushik.Sinha@wichita.edu

→ Code

◆ <https://github.com/rithram/rrkdt>