KD-Tree Algorithm

Behrooz Azarkhalili[1]

[1]Life Language Processing Lab, University of California, Berkeley
[1]azarkhalili@behrooz.ai

Contents

Preprint. Under review.

1 Definition and Overview

A KD-tree (short for k-dimensional tree) is a space-partitioning data structure used for organizing points in a k-dimensional space. KD-trees are useful for various applications, such as range searches, nearest neighbor searches, and other multidimensional search queries. Below, I'll explain the KD-tree algorithm in detail, including its construction, searching, and applications.

2 KD-Tree Construction Explained

2.1 Step 1: Choosing the Axis and Median

Axis Selection:

- **Round-Robin Method:** Traditionally, the axis for splitting the data points in a KD-Tree is chosen in a round-robin fashion across all dimensions (axes) of the data points. This means if the points are in k-dimensional space, the axes are cycled through repeatedly: 0, 1, ..., $k-1$, and then back to 0, and so forth.
- **Alternative Methods:**
  - **Largest Variance Method:** Choose the axis with the greatest variance among the data points. This strategy aims to partition the data along the dimension that exhibits the greatest spread. The variance σ^2 along each axis d is calculated, and the axis with the highest variance is selected:

    $$\sigma_d^2 = \frac{1}{n}\sum_{i=1}^{n}(x_{id} - \bar{x}_d)^2$$

    where x_{id} represents the d-th coordinate of the i-th data point, $\bar{x}_d$ is the mean of all the d-th coordinates, and n is the number of data points.

    $$d^* = \arg\max_d(\sigma_d^2)$$

  - **Range-Based Method:** In this method, the axis with the largest range (difference between maximum and minimum values) among the data points is chosen. This method aims to split along the dimension where the data is most extended, potentially leading to a more efficient partitioning:

    $$\text{Range}_d = \max_i(\{x_{id}\}) - \min_i(\{x_{id}\})$$

    where x_{id} represents the d-th coordinate of the i-th data point.

    $$d^* = \arg\max_d(\text{Range}_d)$$

Median Selection:

- The median is selected to split the data set into two roughly equal parts along the chosen axis. This choice ensures that the tree remains as balanced as possible, preventing skewed distributions of nodes that can lead to inefficient queries.
- Mathematically, the median is chosen as the middle value when the data points are sorted according to their values along the current axis. If there is an even number of points, any central value can be chosen, or the average of the two central values is used.

2.2 Step 2: Recursion

Recursive Partitioning:

- Once the median on the chosen axis is used to split the data into two sets, the same process (step 1) is recursively applied to each subset.
- The recursion proceeds by selecting the next axis in the sequence and finding the median of the subsets to create further splits.

Binary Tree $d=1$ $D=\{(x_1,y_1),\ldots,(x_n,y_n)\}$

$\{x_1,\ldots,x_n\} \rightarrow$ median

$d=2$ $D=\{(2,3),(5,4),(9,6),(4,7),(8,1),(7,2),(1,5),(6,8)\}$

1) we first choose the splitting dimension
2) split database based on the chosen dimension.

1) choose the dimension with the highest Variance
2) choose the dimension with the largest Range
3) Iterate progressively through dimension
   $1 \rightarrow 2 \rightarrow 3$
4) choose it randomly

Approximation Algorithms

Mean (avg):

$(\text{mean})_x = \{2,5,9,4,8,7,1,6\} = \left(\frac{42}{8}\right) = 5.25$

$(\text{mean})_y = \{3,4,6,7,1,2,5,8\} = \left(\frac{36}{8}\right) = 4.5$

$(\text{Range})_x = |9-1| = 8$ ($\max_x$, $\min_x$) $\quad \text{Range}_c = \binom{8}{7}$

$(\text{Range})_y = |8-1| = 7$ ($\max_y$, $\min_y$)

$$(\text{Variance})_x = \frac{1}{n}\sum_{i=1}^{n}(x-\text{mean})^2$$

$$(\text{Variance})_y = \frac{1}{n}\sum_{i=1}^{n}(y-\text{mean})^2$$

- The recursive splitting continues until the subsets cannot be divided anymore, typically when a subset has fewer than two points.

This detailed explanation now provides a comprehensive overview of the different methods used for selecting the axis in KD-Tree construction, with explicit clarifications of each variable used in the computations. Each method aims to optimize the tree structure for efficient spatial querying and storage, accommodating various types of data distributions and dimensions.

3 Searching in a KD-Tree

3.1 Nearest Neighbor Search

The goal of the nearest neighbor search is to find the closest point in the KD-tree to a given query point. The search algorithm involves recursively traversing the tree, pruning branches that cannot contain a closer point than the best one found so far.

- **traversing the Tree:** To find the nearest neighbor of a query point, the tree is traversed starting from the root. At each node, depending on whether the query point's coordinate (for the node's splitting dimension) is less than or greater than the split value, it moves to the left or right child, respectively.
- **Backtrack:** Once a leaf node is reached, the algorithm backtracks, checking at each node if the other subtree could contain a closer point. This involves checking if the distance from the query point to the splitting hyperplane is less than the best found distance to a neighbor.

Detailed Search in a KD-Tree with 2D Sample Points

Sample Data Points

Let's consider the following set of 2D points to construct our KD-Tree:

$$S = \{(2,3), (5,4), (9,6), (4,7), (8,1), (7,2), (1,5), (6,8)\}$$

We will also define a query point $Q = (7,5)$ for which we want to find the nearest neighbor.

Constructing the KD-Tree

We will construct the KD-Tree in a step-by-step fashion. For simplicity and balance, we choose the splitting axis in a round-robin fashion starting from the x-axis (dimension 0) and then the y-axis (dimension 1), and repeat.

Step 1: Root Node (Split on x-axis)

- Sort the points by x-coordinate: $\{(1,5), (2,3), (4,7), (5,4), (6,8), (7,2), (8,1), (9,6)\}$
- The median by x-coordinate is at the fourth position for the root split: $(5,4)$
- This partitions the points into:
  - Left subtree of $(5,4)$: $\{(1,5), (2,3), (4,7)\}$
  - Right subtree of $(5,4)$: $\{(6,8), (7,2), (8,1), (9,6)\}$

Step 2: Left Subtree (Split on y-axis)

- Sort left subtree points by y-coordinate: $\{(2,3), (1,5), (4,7)\}$
- The median by y-coordinate is at the second position: $(1,5)$
- Left subtree of (1, 5): $\{(2,3)\}$
- Right subtree of (1, 5): $\{(4,7)\}$

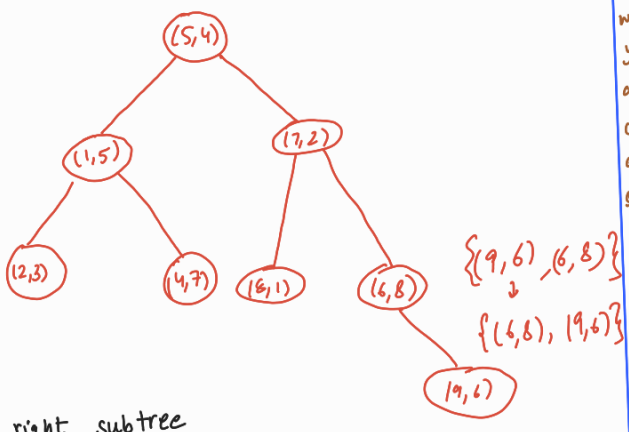
Step 3: Right Subtree (Split on y-axis)

Step 1: [P-3]

Sort by x-axis :-

{(1,5), (2,3), (4,7), (5,4), (6,8), (7,2), (8,1), (9,6)}

root



Left and right subtree

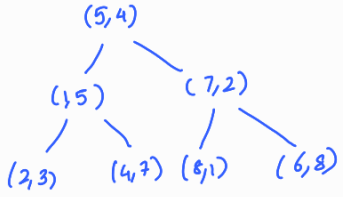
are sorted based on y-axis

How to find K-NN ??

Q = (7,5)

we use priority Queue

or max heap.



1-NN - 2 Steps - 1) traverse (include leaf nodes)

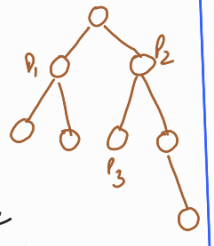
2) 1-NN = {C1, C2, . . . , Cn} [Backtrack, → start from leaf-node
go one step up → check the opp.
subtree do the process till level 1]

For K-NN we follow the same process & keep the candidates in priority queue, and we need to check the other side too.

---

1) NN-point
2) Candidate Point
3) at least one-point → except (Query-point)

1) Sphere Intersection
↳ closest one's
Point within ε-dist.



Query Point Q

$d(Q,P) \leq \varepsilon_0$

In case (3) is true
no need to backtrack

1) Traverse → 1) x-axis → splitting axis
2) Backtrack 2) Root
3) d(Q, Root) ≤ ε0

if (3) is not true, then traverse & backtrack the tree

Small world - Yahoo - 6 degree of seperation

↓

search engine

Hierarchical

(what happened here??)

---

what happens in Chat GPT ?

You put in a question (which is a query). before everything, there's a concept call self-supervised [not rnn]

consider all PDF-book in Canada are ingested and the model is to be trained and the task is to given one sentence, the model needs to predict the next sentence. when you provide a question (query) is converted into a vector (256-d: embedding) when you provided data you input around 10^{12} vectors so it looks for the NN

How is it improving?

- memory
- RLHF [Reinforcement learning based on human feedback]

- Sort right subtree points by y-coordinate: $\{(8,1),(7,2),(6,8),(9,6)\}$
- The median by y-coordinate is at the second position: $(7,2)$
- Left subtree of (7, 2): $\{(8,1)\}$
- Right subtree of (7, 2): $\{(6,8),(9,6)\}$

Further Splitting Right of (7, 2):

- Split next on the x-axis: $\{(6,8),(9,6)\}$
- The median by x-coordinate is at the first position: $(6,8)$
- Left subtree of (6, 8): None
- Right subtree of (6, 8): $\{(9,6)\}$

KD-Tree Structure

The KD-Tree structured based on the above partitioning is as follows:

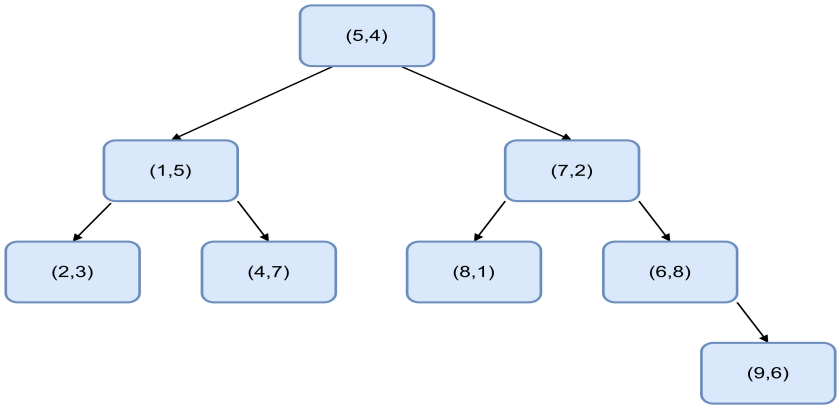


Figure 1: KD-Tree construction

Searching for the Nearest Neighbor

To find the nearest neighbor of the query point $Q = (7,5)$:

1. **Start at the root (5, 4)**, proceed to the right subtree because $x_Q > x_{\text{root}}$.
2. **Move to (7, 2)**, then continue to the right because $y_Q > y_{\text{node}}$.
3. **Move to (6, 8)**, and then continue to the right to (9, 6) because $x_Q > x_{\text{node}}$, which is a leaf node.
4. **Backtrack** to check other branches for closer points considering squared Euclidean distances.

Distance Calculations:

- $d((7,5),(9,6)) = \sqrt{(7-9)^2 + (5-6)^2} = \sqrt{5}$
- Continuing backtracking and comparing, ensuring all regions that could potentially contain a closer point are checked.

The detailed steps show the nearest neighbor search involving comparing distances and backtracking to ensure no closer points are missed due to partitions. This methodical approach ensures accurate nearest neighbor identification in a KD-Tree.

Backtracking in KD-Tree Nearest Neighbor Search

When searching for the nearest neighbor of a query point in a KD-Tree, the algorithm initially descends the tree from the root, moving left or right at each node depending on whether the query point's corresponding coordinate is less than or greater than the node's splitting value. This phase is straightforward; however, it doesn't guarantee that the closest point has been found unless the entire space where the closest point could potentially lie has been explored. This necessity leads to the backtracking step.

Initial Descent

Suppose the query point is $Q = (7, 5)$. The initial descent in the tree might lead us to a leaf node or a node close to the leaf without necessarily being the nearest neighbor. For instance, in the previously discussed tree structure, the initial descent ends at the node $(9, 6)$, assuming it followed the path determined by splitting dimensions and median values.

Backtracking Process

1. **Check Current Best Distance:** When the search reaches a node, say $(9, 6)$, it calculates the distance to the query point. Let's assume this distance is currently the shortest found.
2. **Reverse Path:** The search algorithm then retraces its path up the tree, revisiting nodes it passed during the descent. At each node, it evaluates whether the other subtree (the one not taken initially) could contain a closer point than the current best distance.
3. **Sphere-Plane Intersection:** At each node, the algorithm checks if the hypersphere (circle in 2D, sphere in 3D, etc.) centered at the query point with a radius equal to the current shortest distance intersects the splitting hyperplane of the node. If it does, there is a possibility that the other subtree might contain a nearer point, and thus, that subtree must be searched.

   Mathematically, if the splitting dimension at node N is d, and the splitting coordinate is x_d, then the subtree is searched if:

   $$|x_d - Q_d| < \text{current shortest distance}$$

   where Q_d is the d-th coordinate of the query point Q.
4. **Recursive Searching of Subtrees:** If the sphere-plane intersection test is positive, the subtree is searched recursively in the same manner as the initial descent, potentially updating the best known distance if a closer point is found.
5. **Termination:** Backtracking and subtree searching continue until all nodes that could possibly contain a closer point than the current best known are searched. The algorithm terminates when it returns to the root node and no further subtrees need to be searched.

Example Applied to $Q = (7, 5)$

After initially finding $(9, 6)$ and computing the distance, the search would backtrack to $(6, 8)$. It would check if the left subtree of $(6, 8)$ (which does not exist in this case) needs to be searched. Then, it would backtrack to $(7, 2)$ and check if the left subtree containing $(8, 1)$ could contain a closer point. This decision would be based on the calculated distance at each step.

By expanding the explanation of backtracking, we can understand the importance of this step in ensuring the accuracy and completeness of the nearest neighbor search in KD-Trees. This process effectively handles the complexity of spatial data structures, ensuring that no potential nearest neighbors are overlooked.

4 Recurrence Relation for the Height of a KD-Tree

A KD-Tree splits the data set into two halves at each node, assuming the data can be perfectly divided each time. This recursive division halts when there is only one data point per node, defining the base case of the recursion. Let $h(n)$ represent the height of the KD-Tree with n data points:

- At each level of recursion, the dataset is split into two roughly equal halves.

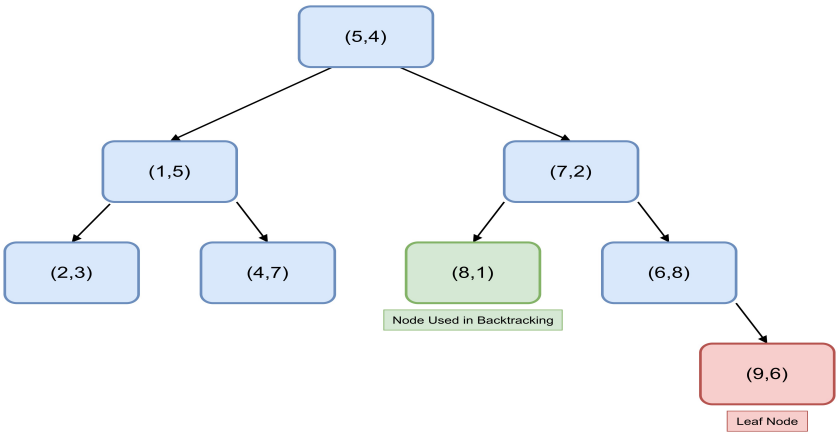


Figure 2: KD-Tree Search

- The height of the tree increases by 1 for each level of split.

This can be represented by the recurrence relation:

$$h(n) = h\left(\frac{n}{2}\right) + 1$$

Here:

- The recursion continues until a single element remains, where $h(1) = 0$ (since a single element or an empty tree does not contribute to the height).

4.1 Computation of KD-Tree Height

Thus, solving this recurrence directly (or by using iterative substitution, which would mimic a Master Theorem-like approach in simplicity), we find:

$$h(n) = \lceil \log_2 n \rceil$$

This solution matches our intuitive understanding that each split divides the dataset into two parts, and the depth of recursive splitting directly corresponds to the binary logarithm of the number of elements.

4.2 Interpretation

The height $h(n) = \log_2 n$ indicates that the KD-Tree, if balanced, has a logarithmic height. This efficient height contributes to the overall logarithmic complexity of search operations in the tree, making KD-Trees effective for multidimensional searching tasks in balanced scenarios. The logarithmic height ensures that operations such as insertion, deletion, and searching can all be performed in $O(\log n)$ time in a well-maintained, balanced KD-Tree.

5 Time Complexity of Constructing and Searching in KD-Trees

KD-Trees are widely used for organizing points in a k-dimensional space, optimizing several operations such as nearest neighbor searches. Understanding the time complexity of constructing and searching these trees is crucial for evaluating their performance in practical scenarios.

Operation	Average	Worst Case
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Table 1: Time Complexity in Big $\mathcal{O}$ Notation

5.1 Time Complexity of Searching in a KD-Tree

6 Applications of KD-Trees

- **Nearest Neighbor Search:** Quickly find the closest point in a dataset to a given query point, useful in pattern recognition, machine learning, and spatial databases.
- **Range Search:** Find all points within a certain range or region, useful in geographical information systems (GIS) and computer graphics.
- **Clustering:** Efficiently cluster points in space by using KD-trees for nearest neighbor queries and range searches.

7 Advantages and Disadvantages

7.1 Advantages

- Efficient searching in multidimensional space.
- Reduces the complexity of nearest neighbor search from $O(n)$ to $O(\log n)$ on average for balanced trees.

7.2 Disadvantages

- Performance can degrade with high-dimensional data due to the "curse of dimensionality."
- Tree construction can be computationally intensive, especially for large datasets.