COSC 520 Assignment 2 - Advanced Data Structures: KD Tree

ZHENBANG HE, Uniersity of British Columbia, Canada

In this report, we introduce the KDTree data structure and explain its basic idea, algorithm steps, and common applications. We implemented a 2D version of KDTree in Python, together with a data generation script. Several versions were developed with different optimization strategies, such as variance-based dimension selection, alternating dimension selection, and approximate median sampling. We tested these versions on generated datasets and compared their building time, query time, and memory usage. The results show that KDTree can greatly improve the efficiency of nearest neighbor search and provide strong acceleration for spatial queries. Through this experiment, we also gained a better understanding of how design choices and optimizations affect the performance of KDTree in practice. The code and dataset are available at https://github.com/m-iDev-0792/COSC520-Assignments/tree/main/Assignment2.

Additional Key Words and Phrases: Algorithm analysis, Algorithm design, Computational theory

ACM Reference Format:
Zhenbang He. 2025. COSC 520 Assignment 2 - Advanced Data Structures: KD Tree. 1, 1 (October 2025), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In many computer science applications, we often need to find the nearest data points in a high-dimensional space. This kind of problem appears in areas such as computer graphics, machine learning, data mining. For example, when we need to find similar images in feature space, detect nearby objects, or search for the closest location on a map, we need an efficient way to perform spatial queries. A simple brute-force method checks all points/vectors one by one, but this becomes very slow when the dataset is large. Therefore, different spatial data structures have been developed to make this process faster.

Among these data structures, the KDTree (k-dimensional tree) is one of the most widely used and classic methods. KDTree organizes data in a binary tree form, where each node splits the space into two parts according to a chosen dimension and a threshold value. This allows the algorithm to quickly narrow down the search area and skip many unnecessary comparisons. KDTree is especially efficient for low or medium-dimensional data, and it is easy to implement and understand compared to more complex structures like Ball Tree or R-Tree.

In this report, we choose KDTree as our topic because it is both conceptually simple and practically useful. By implementing a 2D KDTree in Python, we aim to understand how it works internally and how different optimization strategies can affect its performance. We design and compare several versions with different methods of selecting the split dimension and approximate median sampling. Through experiments on generated datasets, we analyze their building time, query time, and memory usage. This helps us not only to evaluate the efficiency of KDTree but also to learn more about the trade-offs in spatial data structure design.

Author's Contact Information: Zhenbang He, zbhe96@student.ubc.ca, Uniersity of British Columbia, Kelowna, British Columbia, Canada.

ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 KD Tree Algorithm

2.1 Principles and Brief Introduction

KDTree, or k-dimensional tree, is a kind of data structure used to organize points in a k-dimensional space. It was first introduced by Jon Louis Bentley in 1975. The main idea behind KDTree is to divide the space step by step into smaller regions, so that we can search for data points more efficiently. Bentley's original purpose was to create a simple and effective structure for handling multidimensional search problems, such as finding points within a region or finding the nearest neighbor of a given query point. Before KDTree was invented, most algorithms for searching multidimensional data had to check all points one by one, which was very time-consuming. KDTree made a big improvement by using a recursive partitioning strategy, similar to how binary search works in one-dimensional data. Since then, KDTree has become one of the most classic and popular spatial data structures. It is widely used in computer graphics, machine learning (for example in k-nearest neighbor algorithms), robotics (for path planning), and many other fields.

2.2 Algorithm and Construction

A KDTree is a binary tree where each node represents a region of the space. Each node stores a point and a splitting dimension. The space is divided by a hyperplane that is perpendicular to the chosen dimension. The left subtree contains all points that are smaller than the node's value on that dimension, and the right subtree contains all points that are greater.

2.2.1 Construction Process.

(1) **Choose a dimension to split.** There are different strategies to decide which dimension to use. The simplest method is to alternate the dimension at each level (for example, x, then y, then x again in 2D). Another method is to choose the dimension with the highest variance, because that usually provides a better balance and reduces the depth of the tree.
(2) **Find a splitting point.** Usually we select the median value of the chosen dimension, so that the tree remains roughly balanced. This helps keep the search efficient.
(3) **Recursively build subtrees.** The points on the left side of the split go into the left subtree, and the points on the right go into the right subtree. The recursion stops when there are no more points to split.

2.2.2 Query Process. To find the nearest neighbor of a given point, KDTree uses a recursive search.

(1) We start from the root and move down the tree by comparing the query point with the node's splitting dimension, just like in binary search.
(2) When we reach a leaf node, we record the best (closest) point found so far.
(3) Then we go back up the tree and check whether the other side of the split might contain a closer point. If the distance between the query point and the splitting plane is smaller than the current best distance, we need to explore that other branch.
(4) This process is called pruning, because we skip the branches that are impossible to contain a better answer. With pruning, the search becomes much faster than checking all points.

2.2.3 Time Complexity. The time complexity of KDTree depends on the data distribution and the dimension.

- For building the tree, if we use an efficient algorithm to find the median at each step, the time complexity is about $O(N \cdot logN)$, where N is the number of points.
- For searching (nearest neighbor query), the average time complexity is about $O(logN)$, which is much faster than the linear search $O(N)$. However, in the worst case (for example,

when the data is very high-dimensional or unevenly distributed), the search may degrade close to $O(N)$.

In practice, KDTree performs very well for low and medium dimensions (usually fewer than 20). But when the dimension is too high, the performance advantage becomes smaller because of the "curse of dimensionality."

2.2.4 Space Complexity. The space complexity of KDTree is mainly determined by the number of nodes and the extra information stored in each node. For each point, we store the coordinates, the splitting dimension, and links to its child nodes. Therefore, the space complexity is roughly $O(N)$, which is quite efficient. However, if we want to store additional data for each node (for example, bounding boxes, precomputed distances, or cached values), the memory usage can increase. In most simple implementations, KDTree remains very lightweight. For large datasets, memory usage is still much smaller than storing all pairwise distances, which makes KDTree a practical choice for nearest neighbor search.

3 Implementation

In this section, we describe the implementation of our KDTree system and its supporting scripts. The implementation consists of three main components: the KDTree construction and query module (`kd_tree_2d.py`), the dataset generation module (`dataset_gen.py`), and the dataset loading module (`dataset_load.py`).

3.1 Dataset Generation

The dataset generation process is implemented in `dataset_gen.py`. The script can generate any number of random 2D points within a user-defined rectangular area. Points are uniformly sampled using NumPy's random number generator, and the results are written to plain text files in chunks. Each line in a file represents a single data point in the format "x y". The generation process supports parallelism through Python's `multiprocessing` library, allowing large datasets (for example, tens of millions of points) to be created efficiently on multi-core systems.

Several parameters can be configured through command-line arguments, including the number of points, the output directory, the number of workers, and the floating-point precision. The use of fixed random seeds ensures reproducibility across runs. This dataset generator was used to create various test sets for benchmarking different KDTree configurations.

3.2 Dataset Loading

To read the generated datasets, we implemented `dataset_load.py`, which provides a convenient interface for streaming or loading 2D points from text files. The function `load_points()` can load a specified number of points or all points from a given folder. It also supports shuffling and conversion to NumPy arrays for efficient numerical processing. Files are read in buffered mode to improve I/O performance, and both comma- and space-separated formats are supported. This flexibility allows the system to easily handle large datasets without loading them entirely into memory at once.

3.3 Median Selection and Tree Construction

A key part of our implementation is the flexibility in median selection. When the `approx_median` flag is set to `False`, the program finds the exact median using NumPy's `argpartition` function, which has $O(n)$ complexity. When `approx_median` is `True`, the program randomly samples a subset of points and computes an approximate median. This approach reduces computation cost for large datasets, trading off a small amount of accuracy.

The KD-tree is built recursively. Each node splits the current set of points into two subsets around the median value of the chosen axis. The recursive process continues until there are no points left to split. During construction, bounding boxes are updated to facilitate later pruning during nearest neighbor queries. This recursive design follows the divide-and-conquer principle and helps improve search efficiency by localizing computations.

3.4 Nearest Neighbor Query

For nearest neighbor queries, the algorithm performs a recursive search starting from the root. At each node, it compares the target point to the splitting hyperplane and explores the most promising branch first. Then, it decides whether to explore the other branch based on the current best distance and the bounding box of the opposite subtree. This pruning strategy significantly reduces the search space, especially for balanced trees.

3.5 Visualization Module

To better understand the behavior of KD-Tree queries, a visualization module was implemented (see `visualize_kd_tree_2d.py`). This tool uses `matplotlib` and `animation` to dynamically illustrate the step-by-step process of nearest neighbor search.

The visualization highlights different regions of the search space as the algorithm progresses. The light blue areas (Figure. 1 (a)) represent regions that are being visited, while the light gray regions (Figure. 1 (b)) indicate pruned subtrees that are skipped due to bounding-box pruning. The query point is shown as a red dot, and the current best nearest neighbor is displayed as a green star. The animation shows how the KD-Tree recursively explores and prunes regions, helping users intuitively understand how pruning improves efficiency.

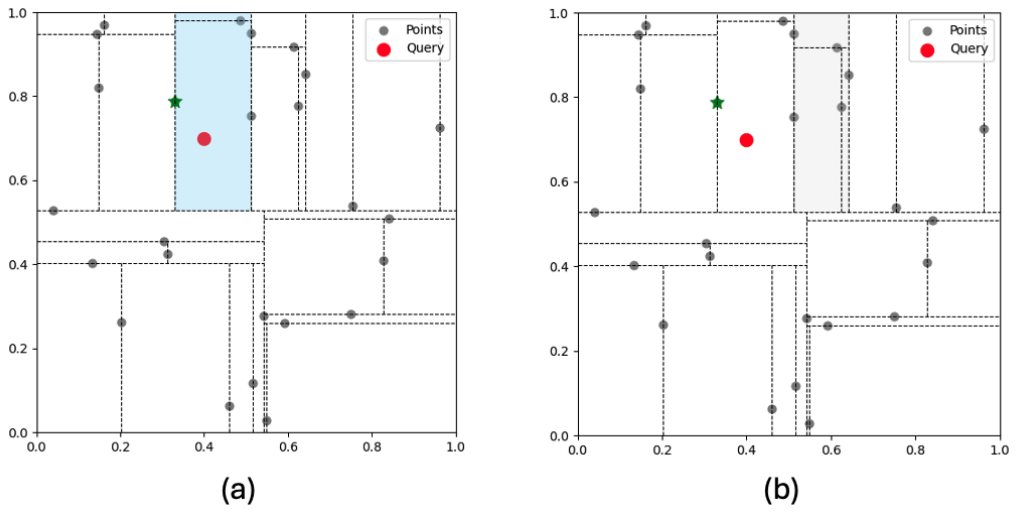

Fig. 1. Visualization of the KD-tree searching process: (a) The light blue region indicates the area currently being visited. (b) The light gray region indicates the area skipped due to pruning.

4 Experiments

4.1 Experimental Setup

To evaluate the performance of the KD-tree implementation, we designed a series of benchmark experiments following the procedure defined in the `benchmark_kdtree()` function. The goal of these experiments is to compare different nearest neighbor search strategies under various dataset sizes, and to understand how construction strategy and splitting rule influence both efficiency and accuracy.

The experiments were performed using randomly sampled 2D point datasets loaded from generated dataset. We split approximately 20% of all points are used as query points and 80% for constructing KD tree. The experiments were conducted on my Mac Mini 2024 desktop, which has an Apple M4 Pro CPU and 64GB RAM.

4.2 Compared Methods

We compared several search strategies implemented in our framework:

(1) **Linear Search (Brute Force):** This method iterates through all points to find the nearest one. Although computationally expensive, it serves as a baseline for correctness and performance reference.
(2) **KD-Tree (Exact Median):** This version builds the KD-tree using the exact median value for partitioning at each recursive step. It ensures balanced partitions but requires additional computation time during construction.
(3) **KD-Tree (Approximate Median):** In this variant, the median is approximated by randomly sampling a fixed number of points (set to 100 in our experiments). This approach speeds up tree construction at the cost of slightly less balanced partitions.

Moreover, for both KD-tree variants, we tested two axis selection strategies:

- **Variance-based:** At each split, the axis with higher variance is chosen to better separate the points spatially.
- **Interleaving-based:** The split alternates between the x and y axes regardless of the data distribution, which is simpler but sometimes less efficient.

By combining these factors, the experiment includes a total of five configurations:

Linear Search, Variance-Exact, Variance-Approximate, Interleave-Exact, Interleave-Approximate.

4.3 Measured Metrics

For each configuration, we measured three main performance metrics:

- **Build Time (s):** The time required to construct the data structure, including median calculation and recursive partitioning.
- **Query Time (s):** The total time for performing nearest neighbor queries for all sampled query points.
- **Peak Memory Usage (MB):** The maximum memory consumption recorded during the process using the `tracemalloc` module.

All measurements were averaged over a single run per dataset size, as the random sampling and approximate median selection already introduce variability. We also ensured reproducibility by fixing the random seed when applicable.

4.4 Data Scale and Repetitions

The benchmarks were conducted across different dataset sizes ranging from 10,000 to 10 million points. Specifically, the tested scales were:

$$N = \{10^4, 5 \times 10^4, 10^5, 2 \times 10^5, 5 \times 10^5, 10^6, 2 \times 10^6, 5 \times 10^6, 10^7\}.$$

For each scale, the query set size was set to **5000**. This large range allows us to observe both small-scale and large-scale performance trends and to examine how different methods scale with increasing data.

4.5 Results and Discussion

The experiment results are shown as Table 1 and the figure drawn from these results are shown as Figure 2.

Build Time. As expected, the linear search method has almost no construction cost since it only stores the raw points. Among KD-tree methods, the exact median construction takes slightly longer, but the difference with the approximate median strategy is small. When the number of points is large, the approximate median strategy does save longer time, but the query performance is also getting worse.

Query Time. The KD-tree consistently outperforms linear search for query efficiency once the dataset exceeds 10^5 points. Both axis selection strategies (variance and interleave) perform similarly for uniformly distributed data, but the variance-based splitting achieves slightly better query times in skewed distributions because it produces more balanced partitions. The approximate median version shows only minor degradation in query performance compared to the exact version.

Memory Usage. Memory consumption increases roughly linearly with dataset size for all methods. The KD-tree requires additional memory to store tree nodes, but this overhead is modest compared to the total data volume. Approximate median construction tends to use slightly less memory due to simpler partitioning operations when the number of points is large($\tilde{2}$million).

Overall Observation. In summary, the experimental results demonstrate that:

- KD-tree is much faster than linear search for large datasets in query phase.
- The approximate median strategy provides an effective trade-off between build speed and query accuracy.
- Variance-based splitting tends to yield more balanced trees and slightly better performance overall.

Figure 2 illustrates the trends of build and query time under different configurations, showing that approximate KD-tree can achieve near-optimal search efficiency with much lower preprocessing time.

In terms of memory, KD-Tree methods required slightly more space than linear search due to additional node and bounding-box structures, but the difference was modest. The gain in query efficiency clearly outweighed the memory overhead.

4.6 Discussion

Overall, the experiments demonstrate the practical benefits of KD-Tree structures for nearest neighbor search in 2D datasets. While linear search remains simple and memory-efficient for very small datasets, the KD-Tree, especially when combined with pruning, offers substantial performance improvements as data size grows. Between the two splitting strategies, variance-based splitting

Table 1. Benchmark results for 2D KD-Tree construction and query performance

N	method	approx_median	sample_size	build_time_s	query_time_s	peak_MB
10000	linear	False	0	0.000688	0.190840	0.160
10000	variance	False	0	3.106273	0.219141	3.012
10000	interleave	False	0	1.342551	0.206843	3.019
10000	variance	True	100	3.054532	0.212815	2.882
10000	interleave	True	100	1.212186	0.210259	3.011
50000	linear	False	0	0.000133	0.799761	0.800
50000	variance	False	0	16.504765	0.288629	15.107
50000	interleave	False	0	6.411401	0.242659	15.106
50000	variance	True	100	16.137145	0.249162	15.064
50000	interleave	True	100	6.318695	0.240600	14.580
100000	linear	False	0	0.000212	1.995282	1.600
100000	variance	False	0	32.012844	0.260243	30.050
100000	interleave	False	0	12.514678	0.251047	30.049
100000	variance	True	100	32.972808	0.274901	29.982
100000	interleave	True	100	12.657161	0.268650	29.854
200000	linear	False	0	0.000561	5.894198	3.200
200000	variance	False	0	70.110231	0.310077	60.224
200000	interleave	False	0	27.576572	0.278487	60.224
200000	variance	True	100	71.131508	0.287372	58.888
200000	interleave	True	100	29.098955	0.286022	60.267
500000	linear	False	0	0.001725	22.412933	8.000
500000	variance	False	0	174.577164	0.296809	149.170
500000	interleave	False	0	66.757586	0.307922	149.170
500000	variance	True	100	170.631111	0.343693	149.728
500000	interleave	True	100	68.032573	0.336161	148.716
1000000	linear	False	0	0.001961	51.440374	16.000
1000000	variance	False	0	361.198375	0.310598	298.463
1000000	interleave	False	0	135.058182	0.336642	298.463
1000000	variance	True	100	343.160351	0.306121	301.846
1000000	interleave	True	100	132.422266	0.305877	301.499
2000000	linear	False	0	0.005522	88.423747	32.000
2000000	variance	False	0	672.744970	0.352974	597.049
2000000	interleave	False	0	258.609322	0.354910	597.050
2000000	variance	True	100	677.693723	0.328770	593.912
2000000	interleave	True	100	272.543088	0.351209	594.096
5000000	linear	False	0	0.013537	250.107056	80.000
5000000	variance	False	0	1683.264676	0.369488	1499.551
5000000	interleave	False	0	659.355386	0.374958	1499.551
5000000	variance	True	100	1680.379310	0.386888	1472.426
5000000	interleave	True	100	673.957171	0.368436	1485.795
10000000	linear	False	0	0.030418	528.382301	160.000
10000000	variance	False	0	3322.518258	0.389814	2999.222
10000000	interleave	False	0	1312.355423	0.641986	2999.222
10000000	variance	True	100	3359.254931	0.649751	2969.507
10000000	interleave	True	100	1360.382877	0.442839	3052.927

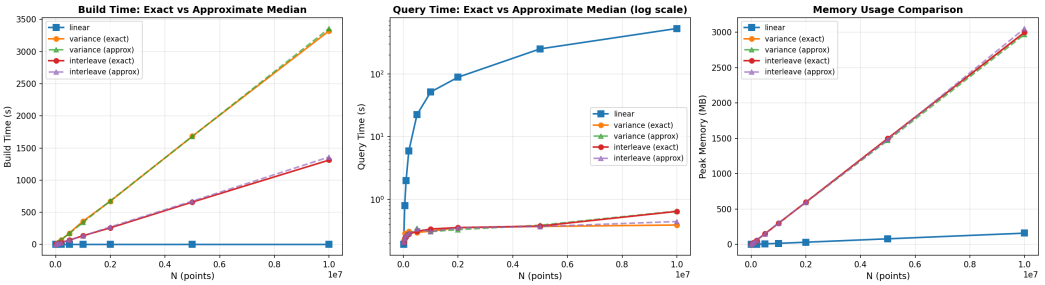

Fig. 2. The build time, query time and memory usage under different configurations.

achieved better balance and lower query times, confirming its effectiveness for data with uneven spatial distributions.

5 Applications of KDTree

KDTree has a wide range of applications in many areas of computer science and engineering. It is mainly used for problems that involve spatial queries, such as finding nearby points, locating objects, or searching in a multidimensional space. Because KDTree can organize data efficiently and reduce the number of comparisons, it is a very useful structure when the dataset is large and the dimension is not too high.

5.1 Typical Application Scenarios

One of the most common applications of KDTree is in **nearest neighbor search**. For example, in machine learning, KDTree is often used in the k-Nearest Neighbor (kNN) algorithm to find the training samples that are closest to a query sample. This can significantly reduce the computation time compared to a brute-force search. In computer vision, KDTree can be used to find matching feature points between images, such as SIFT or ORB features, which helps with image matching and object recognition.

In **robotics and motion planning**, KDTree helps robots find the closest obstacles or paths during navigation. It is also used in **3D graphics and rendering**, where we need to perform ray tracing or spatial partitioning to speed up the rendering process. In **geographic information systems (GIS)**, KDTree can quickly locate nearby cities, stores, or geographical features, making location-based queries much more efficient.

Another interesting use case is in **data compression and clustering**. Because KDTree can easily group points that are close in space, it can be used as a structure to support clustering algorithms such as k-means. It helps speed up the process of finding the nearest cluster center for each point.

5.2 Comparison with Other Methods

Although KDTree is powerful, it is not the only structure for spatial queries. There are several other related methods, each with its own advantages and disadvantages.

- **Ball Tree:** Ball Tree divides the space into hyperspheres instead of hyperplanes. It performs better than KDTree when the data has many dimensions or when the data distribution is very uneven. However, Ball Tree is more complex to implement and usually requires more memory.

- **R-Tree:** R-Tree is another popular structure, mainly used for indexing spatial data such as rectangles or polygons. It is widely used in databases and GIS systems. While KDTree is better for point-based data, R-Tree can handle more complex spatial objects.
- **Brute-Force Search:** This method checks all points directly and has a complexity of $O(N)$ for each query. It is simple but becomes very slow for large datasets. KDTree provides a huge speed improvement when N is large and the dimension is low.
- **Hash-Based Methods:** For very high-dimensional data, sometimes people use approximate methods such as Locality-Sensitive Hashing (LSH). LSH can return approximate nearest neighbors very fast, while KDTree focuses on exact results.

In summary, KDTree is a very good choice when we need exact and efficient nearest neighbor search in low or medium dimensions. It is simple to implement, easy to understand, and performs well in many practical applications. Although it may not be the best choice in very high-dimensional cases, its balance between efficiency and simplicity makes it one of the most widely used data structures for spatial problems.

6 Conclusion

In this report, we studied the KDTree data structure from both theoretical and practical perspectives. We introduced its background, construction method, and different optimization strategies, such as variance-based dimension selection and pruning during search. By implementing several versions of a 2D KDTree in Python and comparing their performance, we were able to clearly observe how KDTree improves the efficiency of nearest neighbor queries compared to a simple brute-force search.

Our experiments showed that KDTree can greatly reduce query time while keeping the memory usage at a reasonable level. The results also indicate that choosing a good split strategy and applying pruning have significant impacts on the performance. Even though the improvement may vary depending on the data distribution, KDTree still provides a solid acceleration method for most low- and medium-dimensional problems.

Through this project, we not only learned how KDTree works internally but also gained a deeper understanding of spatial data structures in general. KDTree is simple yet powerful, and its ideas are closely related to many other algorithms such as Ball Tree, R-Tree, and modern space partitioning methods used in computer graphics and machine learning.

Acknowledgments

The report referred some formulas and algorithms introduced in following links [3–5, 7].Some parts of the code in this assignment were generated with the assistance of the Cursor Editor, a development environment that integrates multiple large language model (LLM) agents. The underlying backends of Cursor may include models such as ChatGPT [6], Claude [1], and others [2]. I hereby acknowledge their contribution. However, since I primarily used the Cursor Editor in a copilot-style fashion, the generated code largely extended my existing code structure and prior snippets. Therefore, the final implementation of this assignment should not be considered as being fully produced by LLMs.

References

[1] Anthropic. 2023. Claude AI. https://www.anthropic.com/claude. Accessed: 2025-10-03.
[2] Cursor. 2023. Cursor: The AI Code Editor. https://cursor.sh. Accessed: 2025-10-03.
[3] GeeksforGeeks. 2023. Ball Tree and KD Tree Algorithms. https://www.geeksforgeeks.org/machine-learning/ball-tree-and-kd-tree-algorithms/ Accessed: 2025-10-25.

[4] GeeksforGeeks. 2023. Search and Insertion in K-Dimensional Tree. https://www.geeksforgeeks.org/dsa/search-and-insertion-in-k-dimensional-tree/ Accessed: 2025-10-25.
[5] Carl Kingsford. n.d.. KD-Trees. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf Lecture notes, Carnegie Mellon University.
[6] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/chatgpt. Accessed: 2025-10-03.
[7] Katyayani Vemula. 2023. What is a K-dimensional Tree? https://medium.com/@katyayanivemula90/what-is-a-k-dimensional-tree-8265cc737d77 Medium article.

Received 20 February 2007