

VNUHCM-UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

CSC10008 – COMPUTER NETWORKING

Challenge 01: KD Tree

Lecturer

Dr. Lê Thanh Tùng

Class

23CLC08

Students

23127071 – Nguyễn Tấn Huy Khôi

23127226 – Nguyễn Đăng Minh

23127255 – Nguyễn Thọ Tài

23127471 – Lê Văn Hoàng Tấn



August 28th, 2024

Contents

1	Enviroments	2
1.1	About	2
1.2	Build	2
1.2.1	CMake - Preferred	2
1.2.2	Script - Only work with window 10+ / Backup methods	2
1.2.3	G++ commandline	2
2	Assignment	3
3	Research	4
3.1	Constructing a KD-Tree	4
3.2	Time complexity of nearest-neighbor searching	5
3.3	Insertion and deletion in KD-Tree	5
3.4	Effect of Dimensionality of data in KD-Tree	5
3.4.1	Performance Impact	5
3.4.2	Curse of Dimensionality:	5
3.5	Arrays versus Linked list implementation in KD-Tree	5
3.5.1	Array implementation:	6
3.5.2	Linked List implementation:	6
3.5.3	Balanced KD-Tree's Limitation	6
3.6	Comparison between KD-Tree and Binary Search Tree	7
4	Implementation	8
4.1	KD-Tree implementation and user interface	8
4.2	Extended Functionality	12
4.2.1	JSON serialization	12
4.2.2	JSON de-serialization & tree reconstruct	12
4.2.3	File Format Justification	13

1 Enviroments

1.1 About

- Language: C++11
- IDE: CLion/Visual Studio
- Build: CMake, compatile with G++
- Dependency: nlohmann/json v3.11.3
- OS: Window (preferred)/Linux

1.2 Build

1.2.1 CMake - Prefered

1. Import the project to any IDE that supports CMake (CLion, Visual Studio,...).
2. Next, build the program as usual. CMake will do everything for you, downloading nlohmann/json and auto-link it with the project.
3. The executable file (kdtree) should apread at the running folder, if not try searching it in cmake output folder (eg: cmake-build-debug,...)

You can also use CMake commandline or CMake GUI to build this project.

1.2.2 Script - Only work with window 10+ / Backup methods

1. There existed a file named **build.cmd**, which is a file that auto compile the code for you using G++. You might need to build nlohmann/json yourself and link it to the project.
2. Run the script. If G++ is not found, install G++. If the build is missing nlohmann/json, compile that yourself then link it to the project.
3. The executable file (kdtree.exe) should apread at the running folder

1.2.3 G++ commandline

If none the aboves worked, you can manually build the project using G++ command. You need to link the dependancy yourself!

```
g++.exe src/main.cpp -o kdtree
```

2 Assignment

Assignment	Subtask	Executor	Progress (%)
Research			
	1	23127071	100%
	2	23127471	100%
	3	23127226	100%
	4	23127226	100%
	5	23127255	100%
	6	23127255	100%
Programming			
	1 - Prepare Dataset	23127471	100%
	2 - Insertion	23127255	100%
	2 - Range Search	23127226	100%
	2 - Nearest Neighbor Search	23127226	100%
	3 - User Interface Option 1	23127226	100%
	3 - User Interface Option 2	23127226	100%
	3 - User Interface Option 3	23127226	100%
	3 - User Interface Option 4	23127071	100%
	3 - User Interface Option 5	23127226	100%
	3 - User Interface Option 6	23127471	100%
	3 - User Interface Option 7	23127255	100%
	3 - User Interface Option 8	23127255	100%
	3 - User Interface Option 9	23127255	100%
	4 - Tree JSON serialization	23127255	100%
	4 - Tree JSON reconstruct	23127255	100%
Report			
	Enviroments	23127255	100%
	Research (1, 2, 3)	23127226	100%
	Research (4, 5, 6)	23127255	100%
	Implementation KD-Tree + interface	23127226/23127255	100%
	Implementation Extended functionality	23127255	100%
	Reference	23127226	100%

3 Research

3.1 Constructing a KD-Tree

Constructing a K-D Tree involves recursively partitioning the points in the space, forming a binary search tree. The process is as follow:

1. Selecting an axis: as we move down the tree, the algorithm alternates between axis in a cycle.

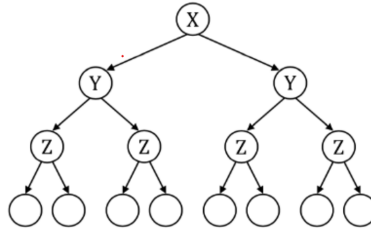


Figure 1: Axis alternation

2. Choosing median and partitioning: along the selected axis, we will choose the middle point (median) to insert into tree. After that, set of remaining points are partitioned into two subsets like binary search tree, as the left subtree will contain points with coordinates less than the split median on the selected axis and vice versa.

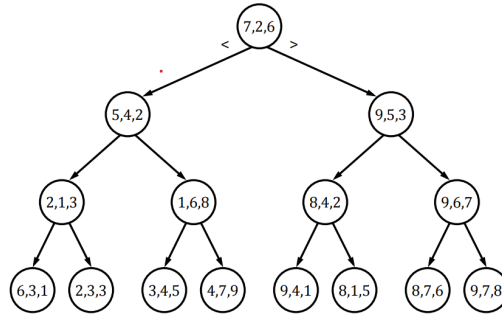


Figure 2: Example of KD tree

How choice of axis and split points affect the tree structure ?

Continuously alternating the axis of K-D tree will ensure that the dimensions are distributed uniformly, avoiding a bias towards any axis. In addition, selecting the median as a split point helps balance the tree as subtrees are created with approximately equal number of nodes (like figure 2). This partition reduces the possibility of a skewed tree and degrades performance. Overall, with a good axis alternation and median selection, a balanced K-D tree is created in which each leaf is approximately the same distance from root.

3.2 Time complexity of nearest-neighbor searching

Since in this project, we construct a K-D tree based on a given set of inputs so pre-processing is made to help the tree balance. Therefore, each level of the tree will eliminate half of the remaining search space, therefore it is similar to balanced binary search tree with logarithmic search time. The balanced property ensures that the searching space is divided as evenly as possible. Tree's height is $O(\log n)$ (n is the input size). [1]

3.3 Insertion and deletion in KD-Tree

Adding a node into KD-Tree is similar to Binary Search Tree as at each level, it will decide which subtree to enter. The plane-slitting property is used for decision. Adding points in this method may causes the tree to become unbalanced, leading to decreased tree performance. If a tree becomes too unbalanced, it may need to be re-balanced for efficient searching operation.[1]

There are two approaches to remove a point. The first one (simple idea and worse time complexity) is to form the set of all nodes from the children of the deleted node and rebuild that part of the tree. The second approach is to find a replacement for the point removed. We'll find the node R that contains the deleted node. For the base case, where R is a leaf node, no replacement is required. For the general case, find a replacement point, called P, from the subtree rooted at R and replace the point stored at R with p. Then, recursively remove p. [3]

3.4 Effect of Dimensionality of data in KD-Tree

3.4.1 Performance Impact

As the number of dimensions (k) increases, the performance of KD-Trees can degrade. The primary reason is that the number of points that need to be examined in nearest neighbor searches or range queries increases exponentially with the number of dimensions.

3.4.2 Curse of Dimensionality:

The "curse of dimensionality" refers to various phenomena that arise when analyzing data in high-dimensional spaces. In the context of KD-Trees, it means that as the number of dimensions increases, the volume of the space increases exponentially, leading to sparsity.

Impact on KD-Trees: For high-dimensional data, the KD-Tree becomes less efficient because the algorithm may need to examine many more nodes than in lower dimensions. As a result, the time complexity can approach $O(n)$, similar to a linear search, making KD-Trees less effective for high-dimensional data.

3.5 Arrays versus Linked list implementation in KD-Tree

Typically, the difference between the two is as follows

3.5.1 Array implementation:

+ Advantages:

- Memory efficiency: since elements are stored in contiguous memories, accessing will cost less than linked list.
- Access time: require less time to access the value in the contiguous memories.

+ Disadvantages:

- Insertion/Deletion complexity: insertion and deletion requires complex tasks to insert and delete node.

3.5.2 Linked List implementation:

+ Advantages:

- Insertion/Deletion complexity: easier to insert or delete nodes since they only involve updating pointers
- Dynamic Size: more flexible memory management, as they can easily grow and shrink.

+ Disadvantages:

- Require more memory: additional memories are needed for storing pointers in each node.
- Access time: slower access times due to pointer de-referencing and lack of memory locality.

3.5.3 Balanced KD-Tree's Limitation

- For an efficient neighbor search and query, the KD tree should be balanced. But balanced kd-tree comes with some limitations. KD-Tree was not designed for that much versatility.

- Since KD Tree has k dimensions, balancing it cannot be done using the method in AVL, red/black tree via rotation as the complexity is huge.

- Insertion/Deletion need to traversal the tree and have a list of all nodes. Any insertion-deletions will be done on this list; this list later got rebuilt back into a balanced tree. This also has high cost of complexity.

3.6 Comparison between KD-Tree and Binary Search Tree

	KD-Tree	
Dimension	k dimension(s)	1 dimension
Construction	Without balancing, it's not much different from BST but comparing each dimensions instead just one. Balanced KD-Tree on other hands required a quick-sort like implementation to build a tree from a given dataset	Simply adding and comparing each node
Search	KD-Trees excel in range searching and nearest neighbor searching in low-dimensional spaces.	Simple traversal to search based on BST properties.
Insert/Delete	More complex than BSTs due to the need for maintaining balance across multiple dimensions. Balanced KD-Tree requires rebuilding the changed tree	Simple BST insertion and deletion
Versatility	No	Yes
Usage	Best suited for applications involving multidimensional and fixed data where range queries or nearest neighbor searches are common, such as spatial databases, computer graphics, and machine learning.	More appropriate for simpler datasets where only 1-dimensional ordering or searching is needed, such as maintaining sorted lists or implementing associative arrays.

4 Implementation

4.1 KD-Tree implementation and user interface

These are structures to manage city data and KD-Tree nodes.

```
struct Data {
    std::string city;
    double latitude, longitude;
};

struct KDTree {
    Data data;
    KDTree* left, * right;
};
```

Figure 3: City and KD-Tree struct

The dataset from Simple Map is preprocessed and filtered. `readCSVFile()` function reads csv file and stores the data in vector.

```
KDTree* readCSVFileIntoTree(const string& filePath) {
    vector<Data> dataset = readCSVFile(filePath);
    if (dataset.empty()) {
        return nullptr;
    }
    return buildKDTree(dataset, 0, dataset.size() - 1);
}
```

Figure 4: Dataset preparation

Insertion is performed as similar as mentioned above. It looks like BST insertion. The depth variable is used to continuously alternate axis at each level.

```
bool insertData(KDTree*& root, Data& data, int depth = 0) {
    if (root == nullptr) {
        root = new KDTree{ data, nullptr, nullptr };
        return true;
    }
    if (depth % 2 == 0) return insertData((data.latitude < root->data.latitude) ? root->left : root->right, data, depth + 1);
    return insertData((data.longitude < root->data.longitude) ? root->left : root->right, data, depth + 1);
}
```

Figure 5: KD-Tree insertion

Nearest-neighbor searching is conducted using the Haversine formula. During the unwinding recursion, we will consider choosing other plane if it has possibility to reduce the distance.

```

void nearestNeighborSearch(KDTree* root, const Data& targ, int depth, bool noCandidate, double& bestDist, Data& bestData) {
    if (root == nullptr) return;
    double dist = getDist(root->data, targ); // haversine formula to calculate distance
    if (noCandidate || dist < bestDist) { // try distance from target location to current node
        bestDist = dist;
        bestData = root->data;
        noCandidate = false;
    }
    if (bestDist == 0) return; // target node is in tree

    double distDim = (depth % 2 == 0 ? root->data.latitude - targ.latitude : root->data.longitude - targ.longitude);
    // distance of current node and target location based on current axis. If distDim > 0 => go to left subtree and vice versa
    (depth += 1) %= 2; // change axis when go to subtree
    nearestNeighborSearch(distDim > 0 ? root->left : root->right, targ, depth, noCandidate, bestDist, bestData);
    if ((long double)distDim * (long double)distDim >= bestDist) return; // No way that remaining dimension has better dist than bestDist
    nearestNeighborSearch(distDim > 0 ? root->right : root->left, targ, depth, noCandidate, bestDist, bestData);
}

```

Figure 6: Find nearest neighbor

Range query function is simple as we will continue going down as long as there are still points in range.

```

bool isInRange(const Data& city, double leftLat, double leftLong, double rightLat, double rightLong) {
    return city.latitude >= leftLat && city.latitude <= rightLat && city.longitude >= leftLong && city.longitude <= rightLong;
}

void rangeQuery(KDTree* root, double leftLat, double leftLong, double rightLat, double rightLong, int depth) {
    if (root == nullptr) return;
    if (isInRange(root->data, leftLat, leftLong, rightLat, rightLong))
        cout << "City (" << root->data.city << ", " << root->data.latitude << ", " << root->data.longitude << ") is in range\n";
    if ((depth % 2 == 0 && root->data.latitude > leftLat) || (depth % 2 == 1 && root->data.longitude > leftLong))
        rangeQuery(root->left, leftLat, leftLong, rightLat, rightLong, depth + 1);
    if ((depth % 2 == 0 && root->data.latitude < rightLat) || (depth % 2 == 1 && root->data.longitude < rightLong))
        rangeQuery(root->right, leftLat, leftLong, rightLat, rightLong, depth + 1);
}

```

Figure 7: Query points in rectangular region

User interface

```

D:\clion\kd-tree>kdtree.exe
-----
Here are your options:
1) Load the list of cities from a CSV file database.
2) Insert a new city into KD-Tree.
3) Insert multiple cities via specified CSV path.
4) Nearest-neighbor search based on giving latitude and longitude.
5) Query cities within a specified rectangular region
6) ===== Quit =====
   ADVANCED FEATURES
7) Print current tree (if exist)
8) Save tree to JSON file
9) Load tree from JSON file
10) Save tree to CSV file
Your option: |

```

Figure 8: Give user all available options

```

Your option: 1
...
Complete loading dataset

```

Option 1 - Load list from simplemap csv database

```

-----
Here are your options:
1) Load the list of cities from a CSV file database.
2) Insert a new city into KD-Tree.
3) Insert multiple cities via specified CSV path.
4) Nearest-neighbor search based on giving latitude and longitude.
5) Query cities within a specified rectangular region
6) ===== Quit =====
    ADVANCED FEATURES
7) Print current tree (if exist)
8) Save tree to JSON file
9) Load tree from JSON file
10) Save tree to CSV file
Your option: 2
City name: meow
Latitude: 12.2
Longitude: -12.2
Insert (meow, 12.2, -12.2) into KD-Tree
-----

```

Option 2 - Add a new city by with given information.

```

Your option: 3
Enter csv file path: test.csv
...
Complete loading csv file
Your option: 7
Tree visualize:
├── meow - (12.2000; -12.2000)
│   ├── cat - (-23.2000; 12.0000)
│   │   ├── São Paulo - (-23.5504; -46.6339)
│   │   └── Jakarta - (-6.2146; 106.8451)
│   └── dog - (12.0000; 111.0000)
└── Manila - (14.5958; 120.9772)
    ├── Mumbai - (18.9667; 72.8333)
    │   ├── Delhi - (28.6600; 77.2300)
    └── Shanghai - (31.1667; 121.4667)
        └── Tokyo - (35.6897; 139.6922)

```

Option 3 - Here we combine option 3 and 7 to easily visualize.

```

Your option: 4
Latitude: 12
Longitude: 12
Closest city to your location is (meow, 12.2000, -12.2000) with distance 2630.3521
Your option: 4
Latitude: -11
Longitude: 23.2
Closest city to your location is (cat, -23.2000, 12.0000) with distance 1802.7496

```

Option 4 - Nearest neighbor searching.

```

Your option: 5
Bottom-left latitude: -5
Bottom-left longitude: 5
Top-right latitude: 5
Top-right longitude: 10
Output csv [Enter to skip]: out.csv
City (São Tomé, 0.3333, 6.7333) is in range
City (Port-Gentil, -0.7167, 8.7833) is in range
City (Bata, 1.8500, 9.7500) is in range
City (Libreville, 0.3901, 9.4544) is in range
City (Buea, 4.1667, 9.2333) is in range
City (Port Harcourt, 4.7500, 7.0000) is in range
City (Malabo, 3.7521, 8.7737) is in range
City (Limbe, 4.0167, 9.2167) is in range
City (Calabar, 4.9500, 8.3250) is in range
City (Douala, 4.0500, 9.7000) is in range
City (Kribi, 2.9500, 9.9167) is in range
City (Nkongsamba, 4.9500, 9.9167) is in range
City (Kumba, 4.6333, 9.4500) is in range
Saved output to file (out.csv)

```

⇒

```

city,lat,lng
São Tomé,0.3333,6.7333
Port-Gentil,-0.7167,8.7833
Bata,1.85,9.75
Libreville,0.3901,9.4544
Buea,4.1667,9.2333
Port Harcourt,4.75,7
Malabo,3.7521,8.7737
Limbe,4.0167,9.2167
Calabar,4.95,8.325
Douala,4.05,9.7
Kribi,2.95,9.9167
Nkongsamba,4.95,9.9167
Kumba,4.6333,9.45

```

Option 5 - Query points in rectangular region. User has option to output to csv file.

```

Here are your options:
1) Load the list of cities from a CSV file database.
2) Insert a new city into KD-Tree.
3) Insert multiple cities via specified CSV path.
4) Nearest-neighbor search based on giving latitude and longitude.
5) Query cities within a specified rectangular region (new database)
6) ===== Quit =====
    ADVANCED FEATURES
7) Print current tree (if exist)
8) Save tree to JSON file
9) Load tree from JSON file
10) Save tree to CSV file
Your option: 8
Output JSON file: file.json
Succeed to save tree to file file.json

Your option: 9
Input JSON file: file.json
Succeed to load tree from file file.json

```

```

0 file.json
1 {
2   "data": {
3     "city": "dog",
4     "latitude": 12.0,
5     "longitude": 111.0
6   },
7   "left": {
8     "data": {
9       "city": "cat",
10      "latitude": -23.2,
11      "longitude": 12.2
12    },
13    "left": null,
14    "right": null
15  },
16  "right": {
17    "data": {
18      "city": "meow",
19      "latitude": 12.2,
20      "longitude": -12.2
21    },
22    "left": null,
23    "right": null
24  }
25 }

```

Option 8 and 9

Option 8 and 9 is used to convert a tree into JSON file format and deserilization the file to reconstruct the KD-Tree.

```

Your option: 10
Output CSV file: out.csv

```

```

city,lat,lng
dog,12,111
cat,-23.2,12
meow,12.2,-12.2

```

mêt

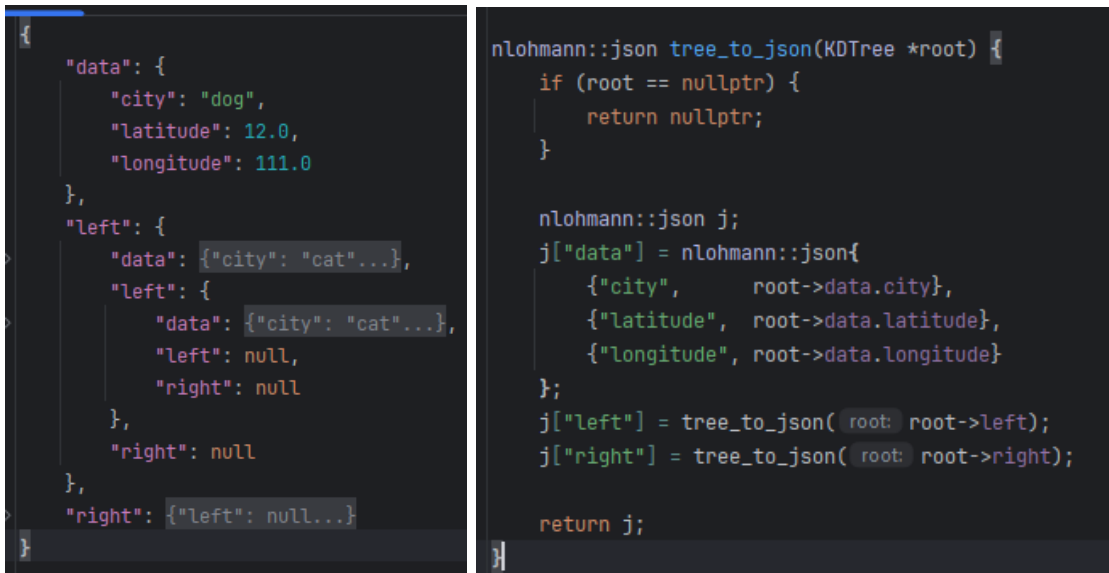
Figure 9: Option 10 - Save tree to a csv file

4.2 Extended Functionality

4.2.1 JSON serialization

Post-order traversing each node then JSON serializing it using nlohmann json[2].

- The data property inside class is a JSON-serialized Node data
- Node's data got parsed 1-1 replica with JSON class.
- Each class has sub classes which is left and right, left and right is the JSON-serialized node. For leaves/nodes with 1 child, the property will be null correspond to the node's missing child.
- The final result got dumped into a file.



```
{
  "data": {
    "city": "dog",
    "latitude": 12.0,
    "longitude": 111.0
  },
  "left": {
    "data": {"city": "cat"...},
    "left": {
      "data": {"city": "cat"...},
      "left": null,
      "right": null
    },
    "right": null
  },
  "right": {"left": null...}
}
```

```
nlohmann::json tree_to_json(KDTree *root) {
    if (root == nullptr) {
        return nullptr;
    }

    nlohmann::json j;
    j["data"] = nlohmann::json{
        {"city", root->data.city},
        {"latitude", root->data.latitude},
        {"longitude", root->data.longitude}
    };
    j["left"] = tree_to_json( root->left);
    j["right"] = tree_to_json( root->right);

    return j;
}
```

4.2.2 JSON de-serialization & tree reconstruct

Post-order traversing each class then JSON de-serializing

- The data property inside class got parsed into Node data
- Node's data got parsed 1-1 replica with JSON class's data.
- Post order traversing left and right class, and repeat the progress, skip if child class is null.
- The final result got parsed into a tree.

4.2.3 File Format Justification

- Advantage:

- Visualizing tree structure in a human friendly form, the structure can be read and understood just by reading the file.
- Intuitive and straightforward, data can easily be dumped and parsed.

- Disadvantage:

- Nested data structure: might result in a big cluster of data.
- Storage size: bigger than binary file implementation.

References

- [1] Hristo Hristov. “Introduction to K-D Trees”. In: (). URL: <https://www.baeldung.com/cs/k-d-trees>.
- [2] N Lohmann. *JSON for modern C++*, *JSON for Modern C++ - JSON for Modern C++*. 2022. URL: <https://json.nlohmann.me/>.
- [3] Wikipedia. *k-d tree*. URL: https://en.wikipedia.org/wiki/K-d_tree#Adding_elements.