COMPARATIVE DATA STRUCTURES

Kshitij Duraphe
BU ID: U60743361
SCC: kshitijd

Zifei Lu
BU ID: U73814476
SCC: luzzfy

Timur Zhussnov
BU ID: U73228594
SCC: zatimur

Tsani Rogers
BU ID: U01866982
SCC: tsrogers

Angela Castronuovo
BU ID: U68837053
SCC: angelac5

December 12, 2022

ABSTRACT

This project aims to implement two data structures: an octree and a kd-tree in C++ and compare their performances with four different metrics: generating point clouds, finding a random point, finding all points within a certain radius, and finding k-nearest neighbors. Our results showed that both implementations had comparatively similar performances regarding searching algorithms, but creating a point cloud takes longer for the octree. The code for the project is available at: `https://www.github.com/ksd3/ec504`.

1 INTRODUCTION

Both octrees and kd-trees are special cases of a generalized data structure known as a binary space partition (BSP) tree. BSP trees partition spae by recursively subdividing it into smaller and smaller portions; the portion dimensions are determined by what type of tree is being implemented. In the process of subdividing, a tree data structure is created. Octrees are one way of implementing a BSP tree. In an octree, each node has has children, and each child has 8 children. The 3D space is thus divided into octants. In point-region octrees, once the space being subdivided is sufficiently small, the center of the smallest box is assigned as the box's identifier. This allows for computers to begin processing each box as individual units. The size of the smallest box depends on the application. In our program, we have implemented a point-region octree which allows us to designate a single point as our smallest box. Each node of a kd-tree is an

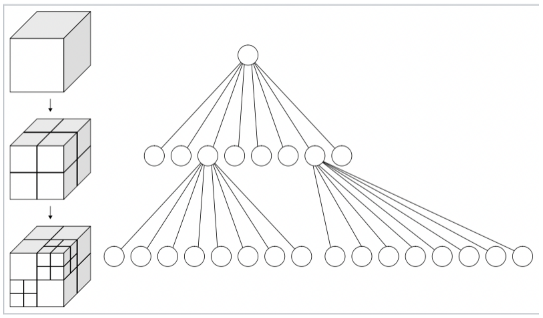

Figure 1: Diagram of an octree

n-dimensional hyperplane which splits the available space into two halves; implicitly a 'left' and a 'right' half. We can recursively split the space along each axis of the kd-tree. This has the advantage of giving us asymmetric dimensions for the smallest bounding boxes, which offers advantages in certain situations such as raytracing.

2 The algorithms

The 4 algorithms we are going to examine here are:

1. Generation of point clouds and creation of trees

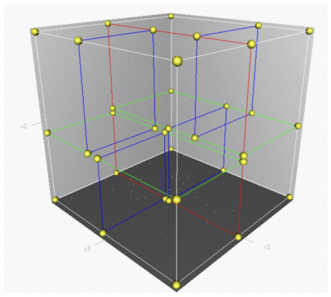

Figure 2: Diagram of a KD-tree

2. Random point search
3. K-nearest neighbors search
4. All-to-all radius search

2.1 Generation of point clouds and creation of trees

We generate points by implementing the `Vec3` data structure made available in C++11 to define a point. A vector of many `Vec3`s is created to hold the points. The points are generated pseudorandomly by using 2022 as the generation seed. They are then recursively inserted into the octree and the kd-tree and timers keep count of the time taken. The size of the trees increase as points are inserted. We observe that point clouds with a low number of total points take almost the same time to create.

2.2 Random point search

We pull a random point from each tree and start from the root of the tree. We then time how long it takes to find this random point.

2.3 K-nearest neighbors search

To find the K-nearest neighbors of a point, we first recursively create a new tree and insert randomly generated points into it. We then pick one point as the base and start to find the k-nearest neighbors. We check which subsection of space (octant or kd-partition) the point belongs to, and calculate the distance to the nearest point in it. We then also check for overlap with other octants and if a closer point is found, the search point is updated. By default, our code finds the 10 nearest neighbors, which then replace the randomly generated points in the new tree. The time complexity of this algorithm depends on the way your tree has been implemented, but $O(n)$ is a decent estimate.

2.4 All-to-all radius search

We take two points in the tree and calculate how long it takes to reach every point within the sphere having the line segment joining these two points as the radius. By default, our starting points are $(10, 10, 10)$ and $(25, 25, 25)$, corresponding to a radius of about 17.

3 Instructions for running the code

Our code is located on the SCC in each of our individual directories as well as on our GitHub profiles. A link to one of them is: `https://www.github.com/ksd3/ec504`. To run this code, simply follow the instructions in the README file. By default, running `make` from a terminal in the repository where you have downloaded this code will suffice, provided that you have C++11 or later. This may require `clang`, but our computers can run it with g++ as well. The `libc++` file is required for the `std:sort` method to run, which is not available by default on the SCC (the SCC uses C++98, which is 24 years old). `std::sort` is required because the `Vec3` implementation of a 3D vector can only be manipulated with `std::sort`. As such, it is highly recommended that you download the source code off the SCC and compile it! You can run `a.out`, which is provided on the SCC; however the improved memory deallocation techniques in C++11 cause the SCC program to leak memory. Running `a.out` on the SCC will still give you a result but it is not the recommended way! If for some reason you cannot run the makefile, running `g++ -std=c++11 -stdlib=libc++`

`*.cpp` in the directory where you have downloaded these files will also work. One of the reasons the makefile may not run is the presence of the /texttt.o files; deleting them and re-running `make` should work.

3.1 Sample Output

A sample output of the code is given below:

```
LNR-D7QDQ6RWF4:forest kshitijd$ g++ -std=c++11 -stdlib=libc++ *.cpp;./a.out
The size of the point cloud that makes up the trees is 50000 points.
Time taken to generate the KD Tree= 0.037463s
Time taken to find one random point=0.002168s
Time taken to find the 10 nearest neighbors is 0.025164s
Time taken to generate the Octree=0.008917s
Time taken to find one random point=0.002003s
Time taken to find the 10 nearest neighbors is 0.019272s
For the KD Tree, all-to-all neighbor finder in the radius 17.3205 took 0.000578s
For the Octree, all-to-all neighbor finder in the radius 17.3205 took 0.00057s
Comparison test passed!
```

For a complete set of output logs, view the `outputs.txt` file in the repository.

4 Sample results and discussion

We generated point clouds for 5,50,500....50000000 points. Beyond that, our program crashed. The graphs of the results for each algorithm are given below.

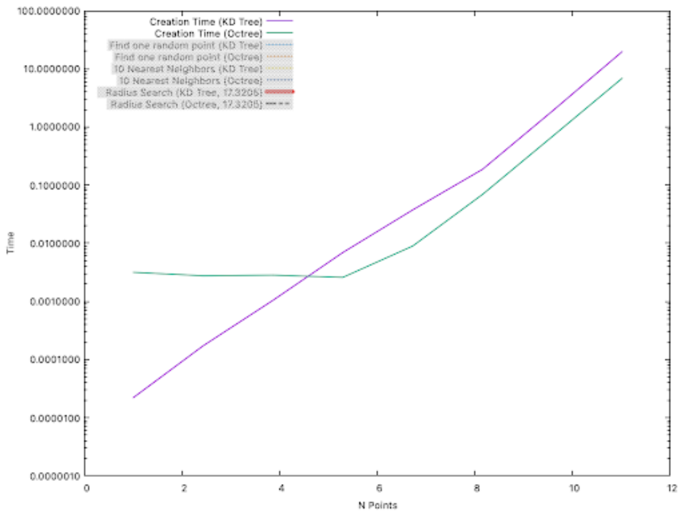

Figure 3: log(point size) vs creation time

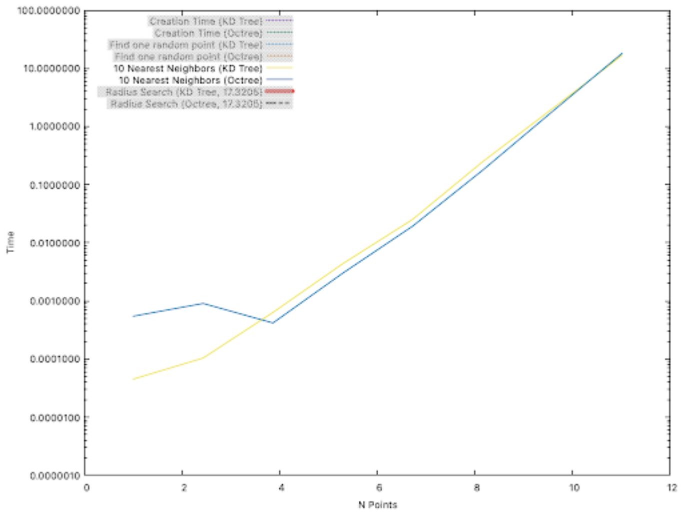

Figure 4: log(point size) vs log(10 nearest neighbors)

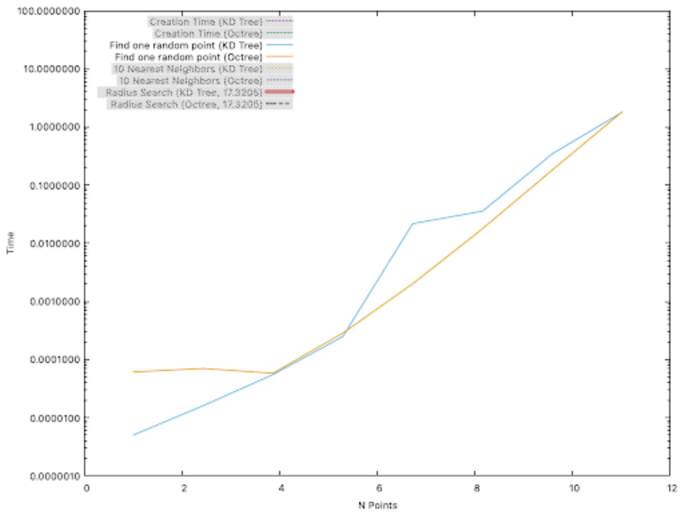

Figure 5: log(point size) vs log(time taken to find one random point)

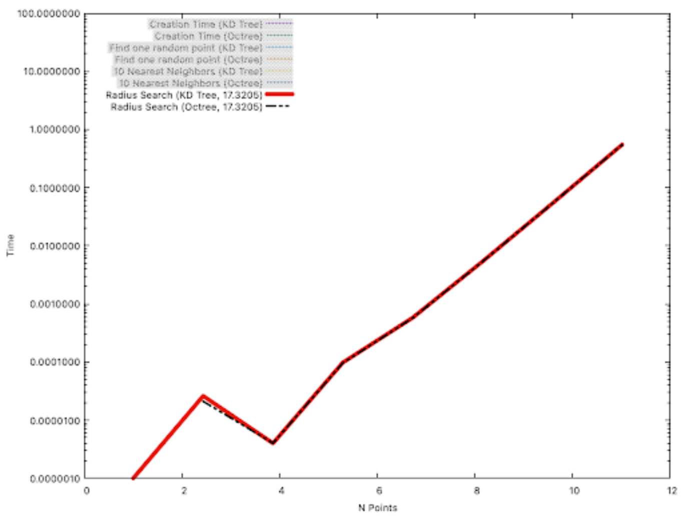

Figure 6: log(point size) vs time taken to search neighbors in a certain radius

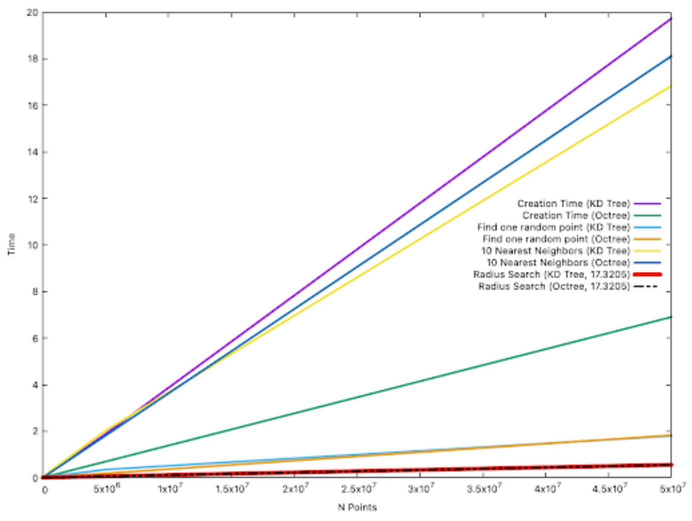

Figure 7: All data

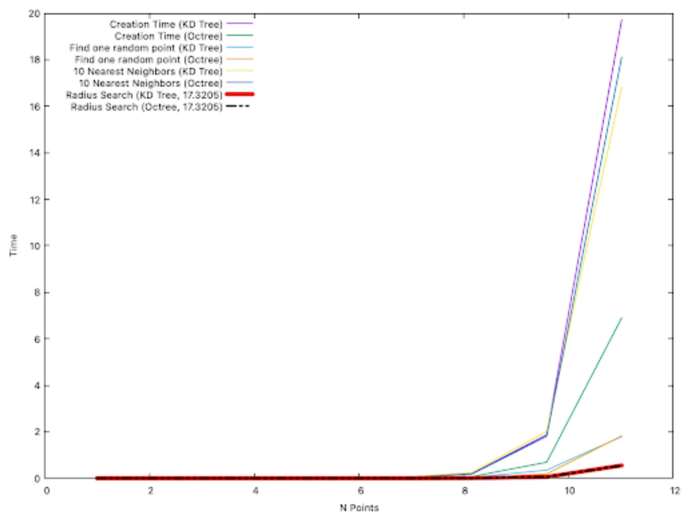

Figure 8: All data plotted with log(points)

We observe that the data structures perform similarly over the majority of algorithms. This may be due to how they were implemented versus how the points lacked features. However we notice that creation of the kd-tree is slightly slower when compared to the octree. This is because the octree's symmetric partitioning of space makes inserting points quicker, compared to the asymmetric partitioning by the kd-tree. Only in the random point search can we truly see a difference, but this is also negligible in real time and can be attributed to the randomness of point selection. Otherwise, our implementation of the data structures perform similarly until 50000000 featureless points.

5 Conclusion

Octrees are easier to create but very slightly harder to run algorithms upon. KD-trees take more time to create but running algorithms upon them is very slightly faster. Selection between the data structures for a particular scenario is task-dependent.

References

[1] Point Cloud Library. [Online]. Available: https://pointclouds.org/.

[2] T. Henning, "Spatial Data Structures: Octrees, BSP, and K-D trees," Observable, 07-Dec-2018. https://observablehq.com/@2talltim/spatial-data-structures-octrees-bsp-and-k-d-trees.