CSI2110 Programming Assignment 2 Report

Matthew Petrucci

300119235

Introduction

This programming assignment is a continuation of the first programming assignment, where clusters are given that represent points in 3D space that reflect a given environment. We must use data structures we make to cluster these points together depending on their distance to one another and how many points are required to be in a given distance to be considered a cluster. The first assignment used a linear stack as the algorithm to classify the points, while assignment 2 aims to improve the run-time of the stack by implementing a KD tree to hopefully reach a minimum of O(log(n)) time, instead of the linear stack O(n) time.

The data shown in this assignment report will almost always be represented as a contrast between both the linear stack implementation and the KD tree implementation. All data can be seen in their raw format in their respective folders.

Experiment 1

Experiment 1 consisted of us given several 3D points and asked to find all neighbours and the amount of neighbours using both a given (and correct) linear stack implementation and our own KD tree implementation. This experiment was mostly to verify that the KD tree implementation worked, and the tests prove that, although both methods work differently, they provide the same results. Here is a short exerpt from the folder. Note, these are from the Exp1_results folder, not exp1.

KD:

```
(-5.429850155, 0.807567048, -0.398216823):
number of neighbors = 5
[(-5.420458778974271,0.7891803562243134,-
0.3973486218703048), (-
5.429850154613408,0.8075670478362598,-
0.3982168226988382), (-
5.43030556398262,0.8246710769927127,-
0.3984338736632657), (-
5.432677820578597,0.8420909833742529,-
0.3987956432309413), (-
5.415942549526783,0.7715622302147948,-
0.3968421613600826)]

(12.97637373, 5.09061138, 0.762238889)
number of neighbors = 2
[(-
12.992860583393504,5.051138148093654,0.762
2934861842156), (-
12.976373725118926,5.090611379773172,0.762
2388885867976)]
```

Linear:

```
(-5.429850155 0.807567048 -0.398216823)
number of neighbors= 5
[(-5.415942549526783,0.7715622302147948,-
0.3968421613600826), (-
5.420458778974271,0.7891803562243134,-
0.3973486218703048), (-
5.429850154613408,0.8075670478362598,-
0.3982168226988382), (-
5.43030556398262,0.8246710769927127,-
0.3984338736632657), (-
5.432677820578597,0.8420909833742529,-
0.3987956432309413)]

(12.97637373 5.09061138 0.762238889)
number of neighbors = 2
[(-
12.992860583393504,5.051138148093654,0.762
2934861842156), (-
12.976373725118926,5.090611379773172,0.762
2388885867976)]
```

Experiment 2

The purpose of this experiment is to display the time difference between using a KD implementation vs a linear implementation experimentally. The results I have obtained show a significant difference in the time taken between a stack and KD tree.

None of the stack results went below 700,000 ns, while the highest KD result was around 85,000 ns. All results can be found in Exp2_results folder.

```
javac *.java && java Experiments exp2 kd
Point_Cloud1.csv 0.5 -5.429850155
0.807567048 -0.398216823
number of neighbors = 459
Average rangequery time for kd at 0.5 eps:
50490

javac *.java && java Experiments exp2 kd
Point_Cloud1.csv 0.5 -12.97637373 5.09061138
0.762238889
number of neighbors = 8
Average rangequery time for kd at 0.5 eps:
68073

javac *.java && java Experiments exp2 kd
Point_Cloud1.csv 0.5 -36.10818686 14.2416184
4.293473762
number of neighbors = 7
Average rangequery time for kd at 0.5 eps:
58237

javac *.java && java Experiments exp2 kd
Point_Cloud1.csv 0.5 3.107437007 0.032869335
0.428397562
number of neighbors = 2106
Average rangequery time for kd at 0.5 eps:
46228

javac *.java && java Experiments exp2 kd
Point_Cloud1.csv 0.5 11.58047393 2.990601868
1.865463342
number of neighbors = 62
Average rangequery time for kd at 0.5 eps:
55462

javac *.java && java Experiments exp2 kd
Point_Cloud1.csv 0.5 14.15982089 4.680702457
-0.133791584
number of neighbors = 30
Average rangequery time for kd at 0.5 eps:
88658
```

```
javac *.java && java Experiments exp2 lin
Point_Cloud1.csv 0.5 -5.429850155
0.807567048 -0.398216823
number of neighbors = 459
Average rangequery time for lin at 0.5 eps:
732354

javac *.java && java Experiments exp2 lin
Point_Cloud1.csv 0.5 -12.97637373 5.09061138
0.762238889
number of neighbors = 8
Average rangequery time for lin at 0.5 eps:
734838

javac *.java && java Experiments exp2 lin
Point_Cloud1.csv 0.5 -36.10818686 14.2416184
4.293473762
number of neighbors = 7
Average rangequery time for lin at 0.5 eps:
760299

javac *.java && java Experiments exp2 lin
Point_Cloud1.csv 0.5 3.107437007 0.032869335
0.428397562
number of neighbors = 2106
Average rangequery time for lin at 0.5 eps:
767307

javac *.java && java Experiments exp2 lin
Point_Cloud1.csv 0.5 11.58047393 2.990601868
1.865463342
number of neighbors = 62
Average rangequery time for lin at 0.5 eps:
733456

javac *.java && java Experiments exp2 lin
Point_Cloud1.csv 0.5 14.15982089 4.680702457
-0.133791584
number of neighbors = 30
Average rangequery time for lin at 0.5 eps:
735265
```

Experiment 3

Experiment 3 was done with two files. Exp3 utilizes the KD method, while Exp3_lin uses the linear stack method. I was surprised at the results and the time difference between the performance of the KD tree and the stack, as I expected the KD tree to totally outperform the stack. However, it seems the time saved in the nearestneighborsKD calculation is somehow dwarfed by the time complexity of the findClusters algorithm that was used in this experiment. This is to the point that the time saved is negigible. We can say the upperbound here to be findClusters, and the lowerbound to be the nearestneighbors calculation in this experiment.

KD:

Point_Cloud1.csv at 2.0 eps and 8 cluster: 350236780

Point_Cloud2.csv at 2.0 eps and 8 cluster: 390417346

Point_Cloud3.csv at 2.0 eps and 8 cluster: 231830165

Linear:

Point_Cloud1.csv at 2.0 eps and 8 cluster: 385434219

Point_Cloud2.csv at 2.0 eps and 8 cluster: 296780600

Point_Cloud3.csv at 2.0 eps and 8 cluster: 283637365

Conclusion

The time saved using a KD tree over a stack is useful for some applications, but insignificant for others. As seen in Experiment 2, a KD tree can significantly outperform a stack with its logarithmic time complexity. However, the time saved on a KD tree implementation vs a linear stack implementation when it comes to real-world applications in algorithms is almost insignificant.

This then asks the question, which one is better? A KD tree will always perform better than a stack for the nearestneighbors algorithm, since at its best it will work logarithmically but at its worst it will perform the same as a stack, being linear. However a KD tree is complicated to understand and complex to implement compared to a stack, and therefore can be harder to debug, secure and maintain than a stack implementation of the algorithm. Is this worth the minuscule amount of time saved, if any time is saved at all? I will say no, and that a stack is a preferable option to a KD tree. It would also depend on the usage of the algorithm. If the algorithm is being performed many many times, the nanoseconds saved in a KD tree implementation may be worth it and save plenty of time. However, if it is being used sparsely such as in our use case, a stack would suffice.

To conclude, these experiments have shown a KD tree to be faster than a stack, but in the grand scheme of things it is not worth the time saved for our use case (running it occasionally). If being used on, say, an autonomous car, a KD tree would be better as it would have to be ran many times a minute.