

The Nearest State County Finder

EC504 2022 FALL

Department of Electrical and Computer Engineering

Team 2		
Names	BU ID	SCC Username
Xin Wang	U56304408	xinw0219
Jianxiao Yang	U77864846	yangjx
Shangzhou Yin	U63027471	syin10
Yuxi Ge	U70788037	yuxig5
Chenjiayi He	U82729958	hcjy

Abstract

To find the number of K (which defined its value is between 1 and 10 in this project) nearest counties of a specific location(target point) entered by the user, the team proposed two possible solutions to solve this problem: K Nearest Neighbors and KD Tree. The dataset for testing this system is collected from the official US Board on Geographic Names, which contains the longitude and latitude of each state and county.

Methodology

K Nearest Neighbors

KNN Theory

First, calculating the distance between the target point to all reference points. Then, building a maxheap to store the number of K closest points, as well as their distances to the target point. Then, We iterate every point and compare its distance to the target point with the maxheap's maximum element. If the maximum element in maxheap is larger, then we pop it out and insert the point we iterate.

Under this case, KNN algorithm needs $O(n \log k)$ to heapify the maxheap in every iteration, so the total time complexity is $O(n \log k)$.

Implementation of KNN

KNN was fully implemented by C++. A Point Class was built and it contained latitude, longitude and county name.

KNN Pseudocode:

```
Input target point's latitude and longitude-> p(la, lo)
Create max-heap to restore the k nearest counties -> result
for county in counties:
    calculate the distance from the county to the target point -> d
    insert the county into max-heap by distance
    heapify the max-heap
    if the size of max-heap > k:
        pop the top one, which is the longest away from the target
        heapify the max-heap

return result
```

Figure 1: KNN Pseudocode

In the command window, enter 'knn' to execute the KNN algorithm. The example is shown in figure 2

```

-search
Enter 'method', 'latitude', 'longitude', 'k'
knn
31
-103
10
0: county: "Pecos" state: "TX"
1: county: "Ward" state: "TX"
2: county: "Crane" state: "TX"
3: county: "Reeves" state: "TX"
4: county: "Winkler" state: "TX"
5: county: "Upton" state: "TX"
6: county: "Ector" state: "TX"
7: county: "Loving" state: "TX"
8: county: "Jeff Davis" state: "TX"
9: county: "Terrell" state: "TX"

```

Figure 2: KNN Pseudocode

K-D Tree

K-D Tree Theory

A K-D Tree (also called a K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space-partitioning data structure for organizing points in a K-Dimensional space. The time complexity of building a K-D Tree is $O(kn \log n)$ since a d-dimensional K-D Tree for a set of n points is a binary tree with n leaves.

Building KD-Tree is to insert points alternating comparison between dimensions. As one moves down the tree, one cycles through the axes used to select the splitting planes.

If we have a target point to which we have to find the k nearest neighbors. Using the tree we made earlier, we traverse through it to find the correct Node. Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes left or right depending on whether the point is lesser than or greater than the current node in the split dimension). Once the algorithm reaches a leaf node, it checks the node point and if the distance is better than the "current best", that node point is saved as the current best and put in a maxheap. The algorithm unwinds the tree's recursion, compares each node with the root of the max-heap, and guarantees that the nodes in min-heap always hold k nearest nodes. When the algorithm finishes this process for the root node, then the search is complete. As we are working in 2 dimensions, the best time will be $\log(n)$. (Binary Heap, 2022)

Implementation of K-D Tree

KD Tree was fully implemented by Java. A Point Class was built, and it contained latitude, longitude, county name, LeftPoint and RightPoint. Besides this, two empty ArrayLists (leftChild, rightChild) were also initialized to contain the new points/counties and were cleaned after building the tree.

```

3 public class Point implements Comparable<Point>{
4     private double x;
5     private double y;
6
7     public String name="undefined";
8     public boolean judge=false; // false -> x , true -> y
9     public int depth;
10    public Point left=null;
11    public Point right=null;
12    public Point father=null;
13    public ArrayList<Point> leftChilds;
14    public ArrayList<Point> rightChilds;

```

Figure 3: Point Class attributes

Three different constructors were provided to initialize the Point Class.

```

15 public Point(double x,double y,String name){
16     this.x=x;
17     this.y=y;
18     this.depth=0;
19     this.leftChilds=new ArrayList<>();
20     this.rightChilds=new ArrayList<>();
21     this.name=name;
22 }
23 public Point(double x,double y){
24     this.x=x;
25     this.y=y;
26     this.depth=0;
27     this.leftChilds=new ArrayList<>();
28     this.rightChilds=new ArrayList<>();
29 }
30 public Point(){
31     this.x=0;
32     this.y=0;
33     this.depth=0;
34     this.leftChilds=new ArrayList<>();
35     this.rightChilds=new ArrayList<>();
36 }
37 public double getX(){
38     return x;
39 }
40 public double getY(){
41     return y;
42 }
43

```

Figure 4: three constructors to initialize the Point Class

Override Point Comparator: use a Boolean type of variable “judge” to determine whether the Point is compared by latitude or longitude. If the “judge” is true, then the latitude will be compared to sort the arrayLists mentioned above, otherwise, the longitude will be compared to sort the same arrayLists.

```

59 | @Override
60 | public int compareTo(Point o) {
61 |     if(judge) {
62 |         if(this.getY()>o.getY()) {return 1;}
63 |         else if(this.getY()==o.getY()) {return 0;}
64 |         else {return -1;}
65 |     }
66 |     else {
67 |         if(this.getX()>o.getX()) {return 1;}
68 |         else if(this.getX()==o.getX()) {return 0;}
69 |         else {return -1;}
70 |     }
71 | }
72 | }

```

Figure 5:override function

Calculate the distance between two points.

```

44 | public double distance(Point other) {
45 |     double x_other=other.getX();
46 |     double y_other=other.getY();
47 |     double R=6371;
48 |     double x1=x*Math.PI/180;
49 |     double x2= x_other*Math.PI/180;
50 |     double y1=y*Math.PI/180;
51 |     double y2=y_other*Math.PI/180;
52 |     double xx=(y1-y2)*Math.cos((x1+x2)/2);
53 |     double yy=x1-x2;
54 |     double distince=R*Math.sqrt(xx*xx+yy*yy);
55 |     return distince;
56 | }

```

Figure 6:calculate the distance between two points

The basic attributes of KD Tree are shown in the following. An insert function is implemented to enter all points.

```

3 | public class KD_tree {
4 |     private ArrayList<Point> points;
5 |     Point root;
6 |
7 |     private int size;
8 |
9 |     public KD_tree(){
10 |         points=new ArrayList<>();
11 |         size=0;
12 |     }
13 |     public int size(){
14 |         return this.size;
15 |     }
16 |     public void insert(Point p) {
17 |         p.judge=false;
18 |         points.add(p);
19 |         size++;
20 |     }

```

Figure 7: KD tree's basic attributes

Then, the tree is constructed by calling init function. It chooses the middle point as the root, and by using the Point Comparator function, put all points that are smaller than middle points to the leftChild and put

all points that are bigger or equal to the middle points to the rightChlds. This tree is similar to the binary search tree, but in this case, the Point Comparator will change in each level of depth.

```

21 |     public void init(){
22 |         Collections.sort(points);
23 |         this.root=points.get(points.size()/2);
24 |         for( Point i:points) {
25 |             if(i!=root) {
26 |                 i.father=root;
27 |                 i.depth++;
28 |                 if(i.compareTo(root)<0) {root.leftChlds.add(i);}
29 |                 else {root.rightChlds.add(i);}
30 |                 i.judge=!root.judge;
31 |             }
32 |         }
33 |         buildTree(root);
34 |     }

```

Figure 8: init function

buildTree function is a recursion function to call getPointChild. getPointChild function is implemented to pick the middle point as the root, and then the points smaller than the root will be put in the leftChild arrayList and the points larger than the root will be put in the rightChild arrayList. To achieve this, we use the 'getPointChlds' function to convert the root and arrays to a root with two childArrays.

```

35 |     //root leftchild rightchild needs to be done
36 |     // to find the leftchild and rightchild
37 |     private void buildTree(Point root){
38 |         if(root.leftChlds.size()>0) {
39 |             Collections.sort(root.leftChlds);
40 |             root.left=root.leftChlds.get(root.leftChlds.size()/2);
41 |             getPointChlds(root.left,root.leftChlds);
42 |             buildTree(root.left);
43 |         }
44 |         if(root.rightChlds.size()>0) {
45 |             Collections.sort(root.rightChlds);
46 |             root.right=root.rightChlds.get(root.rightChlds.size()/2);
47 |             getPointChlds(root.right,root.rightChlds);
48 |             buildTree(root.right);
49 |         }
50 |         root.leftChlds.clear();
51 |         root.rightChlds.clear();
52 |         return;
53 |     }

```

Figure 9: buildTree function

```

79 |     // base needs to be sorted, root needs to be in base
80 |     private void getPointChlds(Point root,ArrayList<Point> base){
81 |         for(Point i:base) {
82 |             if(i!=root) {
83 |                 i.father=root;
84 |                 i.depth++;
85 |                 if(i.compareTo(root)<0) {root.leftChlds.add(i);}
86 |                 else {root.rightChlds.add(i);}
87 |                 i.judge=!root.judge;
88 |             }
89 |         }
90 |     }

```

Figure 10: getPointChild function

The above figures show the steps of building the tree. To find the closest neighbors (K nearest Neighbors function), it is necessary to build up a maxheap (size of K) and use a recursion to find out whether a point should be considered in a maxheap.

```

92     public ArrayList<Point> KNearestNeighbor(int k, Point p) {
93         ArrayList<Point> res = new ArrayList<>();
94         PriorityQueue<Point> maxHeap = new PriorityQueue<>((a, b) -> {
95             double d1 = a.distance(p);
96             double d2 = b.distance(p);
97             if (d1 < d2) { return 1; }
98             else if (d1 == d2) { return 0; }
99             else { return -1; }
100         });
101         // return (int)(d2-d1);
102         Point tmp = this.root;
103
104         recursionSearch(p, tmp, maxHeap, k);
105         while (!maxHeap.isEmpty()) {
106             res.add(maxHeap.poll());
107         }
108         return res;
109     }

```

Figure 11: KNearestNeighbor function

The recursionSearch is based on three steps:

1. call recursionSearch for left/right child
2. compare the target point with the current point
3. check right/left child if they should be considered in the maxheap

The first step should be executed after comparing the latitude/longitude with the target Point. Meanwhile, we can also check whether other child points should be considered by comparing the minimum distance between the current point's plane to the target point with the maxheap's maximum element. Then return to the previous level and repeat the same step.

```

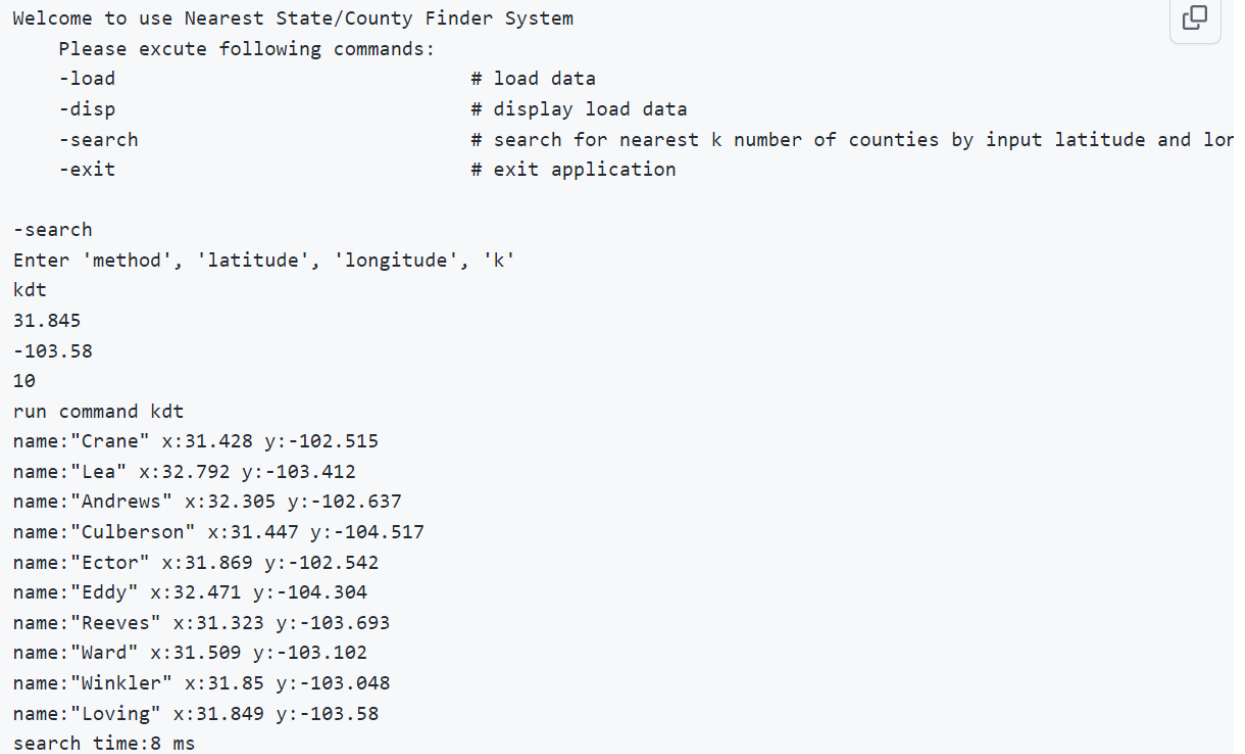
111 private void recursionSearch(Point target, Point cur, PriorityQueue<Point> maxHeap, int k) {
112     if (cur == null) { return; }
113
114     if (cur.compareTo(target) >= 0) {
115         recursionSearch(target, cur.left, maxHeap, k);
116
117         if (cur.right != null) {
118             if (maxHeap.isEmpty() || maxHeap.size() < k) { recursionSearch(target, cur.right, maxHeap, k); }
119             else {
120                 double maxDistance = maxHeap.peek().distance(target);
121                 double targetToRight;
122                 if (!cur.right.judge) {
123                     targetToRight = Math.abs(target.getX() - cur.right.getX());
124                 } else {
125                     targetToRight = Math.abs(target.getY() - cur.right.getY());
126                 }
127                 if (targetToRight < maxDistance || maxHeap.size() < k) {
128                     recursionSearch(target, cur.right, maxHeap, k);
129                 }
130             }
131         }
132     }
133     else {
134         recursionSearch(target, cur.right, maxHeap, k);
135         if (cur.left != null) {
136             if (maxHeap.isEmpty() || maxHeap.size() < k) { recursionSearch(target, cur.left, maxHeap, k); }
137             else {
138                 double maxDistance = maxHeap.peek().distance(target);
139                 double targetToLeft;
140                 if (!cur.left.judge) {
141                     targetToLeft = Math.abs(target.getX() - cur.left.getX());
142                 } else {
143                     targetToLeft = Math.abs(target.getY() - cur.left.getY());
144                 }
145                 if (targetToLeft < maxDistance || maxHeap.size() < k) {
146                     recursionSearch(target, cur.left, maxHeap, k);
147                 }
148             }
149         }
150     }
151     if (maxHeap.size() < k) { maxHeap.add(cur); }
152     else if (cur.distance(target) < maxHeap.peek().distance(target)) {
153         maxHeap.poll();
154         maxHeap.add(cur);
155     }
156 }
157

```

Figure 12: recursionSearch function

Result:

Our command window is executed by Terminal. The average of running time is 8 ms.



```
Welcome to use Nearest State/County Finder System
Please excute following commands:
-load                # load data
-disp               # display load data
-search             # search for nearest k number of counties by input latitude and lon
-exit              # exit application

-search
Enter 'method', 'latitude', 'longitude', 'k'
kdt
31.845
-103.58
10
run command kdt
name:"Crane" x:31.428 y:-102.515
name:"Lea" x:32.792 y:-103.412
name:"Andrews" x:32.305 y:-102.637
name:"Culberson" x:31.447 y:-104.517
name:"Ector" x:31.869 y:-102.542
name:"Eddy" x:32.471 y:-104.304
name:"Reeves" x:31.323 y:-103.693
name:"Ward" x:31.509 y:-103.102
name:"Winkler" x:31.85 y:-103.048
name:"Loving" x:31.849 y:-103.58
search time:8 ms
```

Figure 13: Terminal interface

Reference List

“Binary Heap.” *GeeksforGeeks*, 2 Nov. 2022, <https://www.geeksforgeeks.org/binary-heap/>.

K-nearest Neighbors | *Brilliant Math and Science Wiki*. brilliant.org/wiki/k-nearest-neighbors.