Nearest State/County Finder

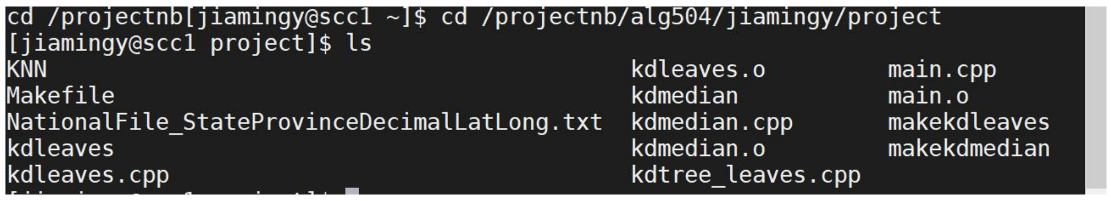
Jiaming Yu
Boston University
jiamingy@bu.edu

Abstract

This project propose to design a nearest state/county finder to return nearest K reference points and the state and county by entering a decimal latitude and longitude. In this project, k-d tree is chosen as our data structure to load all the data extracted from the official US Board on Geographic Names data set as input. In the building process, both of split by median and randomly insert method is implemented and compared. In the querying process, a trick to check the distance between query point and pivot is used to largely improve the efficiency and a method of storing all the data in the leaves is implemented. Therefore, three versions of k-d tree are implemented and compared. Command line user interface results shows that we can use these to query K nearest points and find state and county by voting for 5 nearest points. The result shows that the search of these implementation is efficient.

Code

Project code is stored on SCC /projectnb/alg504/jiamingy/project. Figure 1 shows the list of this dictionary.



```
cd /projectnb[jiamingy@scc1 ~]$ cd /projectnb/alg504/jiamingy/project
[jiamingy@scc1 project]$ ls
KNN                                          kdleaves.o         main.cpp
Makefile                                     kdmedian           main.o
NationalFile_StateProvinceDecimalLatLong.txt  kdmedian.cpp       makekdleaves
kdleaves                                     kdmedian.o         makekdmedian
kdleaves.cpp                                 kdtree_leaves.cpp
```

Figure 1 list of jiamingy/project

There are three versions of kd-tree.

(1) Split by median and store data in leaves

Code is in **kdleaves.cpp**, make file is named makekdleaves and the output is named kdleaves

(2) Split by median and store data in node and leaf

Code is in **kdmedian.cpp**, make file is named makekdmedian and the output is named kdmedian

(3) Randomly insert and store data in node and leaf

Code is in **main.cpp**, make file is named Makefile and the output is named KNN.

1. Introduction

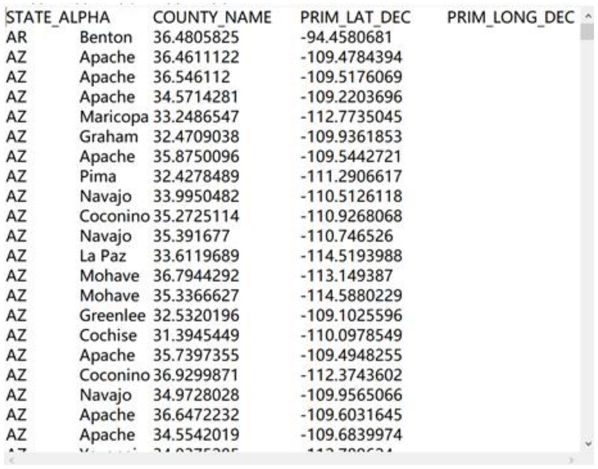
We load the data set including state, county, latitude and longitude and thus build k-d tree data structure to accelerate the query process. In this section, Data set and K-d tree will be introduced.

1.1 Problem Formulation

We are given a large number of reference points, and the system designed would load these reference points into the data structure. Use the data structure to allow users search k nearest points and find the state and county by voting by entering decimal latitude and longitude.

1.2 Data set

The data set we chose is extracted from the official US board on Geographic Names data set. As is shown in the figure 2, the data set contains {state, county, latitude, longitude} as one line in text. And the data set includes more than 2 million points.



```
STATE_ALPHA    COUNTY_NAME    PRIM_LAT_DEC    PRIM_LONG_DEC
AR        Benton    36.4805825    -94.4580681
AZ        Apache    36.4611122    -109.4784394
AZ        Apache    36.546112     -109.5176069
AZ        Apache    34.5714281    -109.2203696
AZ        Maricopa  33.2486547    -112.7735045
AZ        Graham    32.4709038    -109.9361853
AZ        Apache    35.8750096    -109.5442721
AZ        Pima      32.4278489    -111.2906617
AZ        Navajo    33.9950482    -110.5126118
AZ        Coconino  35.2725114    -110.9268068
AZ        Navajo    35.391677     -110.746526
AZ        La Paz    33.6119689    -114.5193988
AZ        Mohave    36.7944292    -113.149387
AZ        Mohave    35.3366627    -114.5880229
AZ        Greenlee  32.5320196    -109.1025596
AZ        Cochise   31.3945449    -110.0978549
AZ        Apache    35.7397355    -109.4948255
AZ        Coconino  36.9299871    -112.3743602
AZ        Navajo    34.9728028    -109.9565066
AZ        Apache    36.6472232    -109.6031645
AZ        Apache    34.5542019    -109.6839974
```

Figure 2: Data set

Although the points with latitude and longitude can be considered as 3-D, since there are so many points and we can use equirectangular approximation to estimate the distance, we can consider it as a 2-D problems.

1.3 K-d tree

K-d tree [1] is short for k-dimensional tree. K represents the number of dimensions, like 2-d tree, 3-d tree. As a special case of binary space partitioning tree, k-d tree is a useful data structure for nearest neighbor search. And therefore, we choose k-d tree to be our data structure.

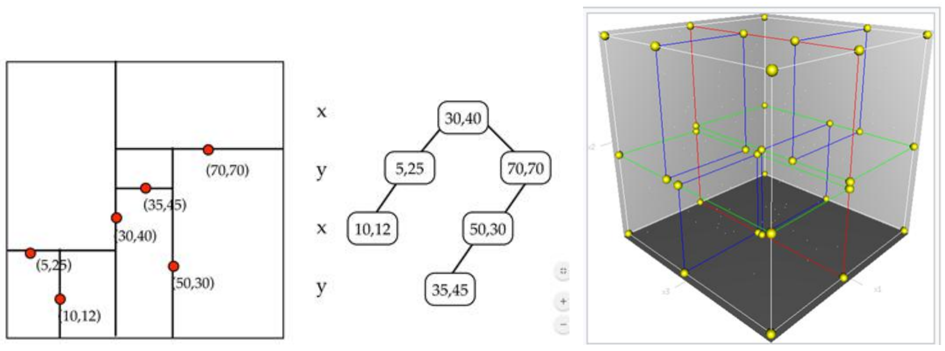


Figure 3: K-d tree

Figure 3 shows the examples of k-d tree. Every non-leaf node can be considered as a split to make the space become two pieces. And the split dimension of each node is related to the level of this node in the tree. That is to say, each level of the trees compare against 1 dimension. Normally, the dimension would be chosen in Round Robin order, i.e. x, y, x, y in 2-D and x, y, z, x, y, z in 3-D. Another idea is to choose the dimension with the largest variance. And therefore, k-d tree can split the space and it can be helpful in nearest neighbor search.
In this project, 2-D tree is used and x, y, x, y would be chosen as split dimension with the depth increases.

2. Approach

In the approach section, we will introduce how we built the tree and use the tree to search for the nearest neighbor points. In the build part, we will discuss two methods: one is split by median and the other is randomly insert. In the query part, we would talk about the method and some tricks.

2.1 Build

2.1.1 Node

To build the k-d tree, we need to decide what should be included in a node structure. The node should includes:
(1) the data (state, county, latitude, longitude)
(2) split dimension
(3) pivot
(4) some bool values like isLeaf, isLeft, isRight to decide the position of the node in the tree which can be helpful in the implementation of nearest neighbor search.
(5) The node also has node *left, *right, *parent so that the data structure can be built as a tree.

2.1.2 Load Data set

The data set is loaded through ifstream and getline() to derive a line of the data set which is also a reference point. Then use vector to split into four parts according to the blank. The string before the first blank would be the state. And the last two string would be latitude and longitude which would be transferred to double by stod(). This design is because county could have several blanks inside it.

The code is shown as follows. All the reference points would be saved inside the struct array which use the struct named Data which has string state, string county and double point[k] where k is defied as 2. Use while() to load all the reference points.

```
Data *inputdata = new Data[2300000];
ifstream fin;
fin.open("NationalFile_StateProvinceDecimalLatLong.txt");
string str;
getline(fin,str);
int N = 0;
vector<string> res;
while (!fin.eof())
{
    getline(fin,str);
    if(str.empty())
        break;
    stringstream input(str);
    string result;
    while(input>>result) {res.push_back(result);}
    int len = res.size();
    inputdata[N].state = res[0];
    inputdata[N].county = res[1];
    if(len>4)
    {
        for(int j=2;j<len-2;j++)
        {
            inputdata[N].county += " "+res[j];
        }
    }
    inputdata[N].point[0] = stod(res[len-2]);
    inputdata[N].point[1] = stod(res[len-1]);
    N = N+1;
    res.clear();
}
fin.close();
```

2.1.3 Split by median

Normally, the k-d tree would be build by choose the median of the split dimension of each level and then add into the tree which will build a balanced tree. Therefore, in the implementation, we choose quick sort to sort the Data struct array we just loaded according to the latitude or the longitude (according to the depth). The smaller would be split to left sub tree and the larger would be split to the right sub tree. The pivot chosen is stored in the node. For example, in the depth of 0, sort all the data by the dimension latitude, suppose a point would sorted to be the median and we would store 0 as split(For split dimension, 0 represents latitude, 1 represents longitude)and store latitude of this point as pivot. Then the data set is divided to left data set and right data set assigned to left sub tree and right sub tree. Along the build process, split, pivot, bool value and so on are assigned. The code is shown in the following. And the time complexity would be $O(n\log^2 n)$

```
struct Node* build(struct Node *root,struct Data* data,unsigned depth,int n)
{
    unsigned split = depth % k; //split dim
    if(n==0)
    {
        root = NULL;
        return root;
    }
    if(n==1)
    {
        root->split = split;
        root->pivot = data->point[split];
        root->isLeaf = true;
        root->left = NULL;
        root->right = NULL;
        root->state = data->state;
        root->county = data->county;
        root->lat = data->point[0];
        root->lon = data->point[1];
        return root;
    }
    root->isLeaf = false;
    data = Sort(data,split,n);
    int mid = (n+1)/2;
    struct Data* Leftdata = new Data [mid];
    struct Data* Rightdata = new Data [n-mid];
    struct Data midData;
    midData = data[mid];

    for(int i=0;i<mid;i++)
    {
        Leftdata[i] = data[i];
    }
    for(int i=0;i<n-mid-1;i++)
    {
        Rightdata[i] = data[mid+i+1];
    }

    root->left = new Node;
```

```
        root->left->parent = root;
        root->left->isLeft = true;
        root->left->isRight = false;

        root->right = new Node;
        root->right->parent = root;
        root->right->isLeft = false;
        root->right->isRight = true;

        root->split = split;
        root->pivot = midData.point[split];
        root->state = midData.state;
        root->county = midData.county;
        root->lat = midData.point[0];
        root->lon = midData.point[1];

        root->left = build(root->left,Leftdata,depth+1,mid);
        root->right = build(root->right,Rightdata,depth+1,n-mid);

        return root;
}
```

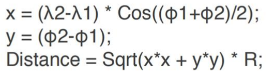
2.1.3 Randomly insert

Randomly insert would cause a unbalanced tree and thus rarely be used to build the k-d tree. Compared to use the median, it just use the reference points in the order of data set. The reason to mention it is because we want to point out that the query for some points under an unbalanced tree can sometimes be better. Since it is an unbalanced tree, some paths to the query point would be shorter while other paths would be longer. The average performance of query for unbalanced tree would be worse than split by median. In the project proposal, (33.24, -112.75) is suggested to be searched and in the implementation of randomly insert, the query would require less time since the path to this point of unbalanced tree would be shorter. The building time of randomly insert would be shorter to be $\mathrm{O}(n \log n)$

2.2 Query

2.2.1 Nearest neighbor search

First we need to define the distance and the distance is defined in the figure 4.



```
x = (λ2-λ1) * Cos((φ1+φ2)/2);
y = (φ2-φ1);
Distance = Sqrt(x*x + y*y) * R;
```

Figure 4: distance function

We then search down to the leaves and then backtrack to the root.
Attention here that we can not just regard it as a BST and only check the points along the path. Consider the example in the following figure 5. The point search down the root and only consider the distance to A and B. However, we also need to consider the distance to C.

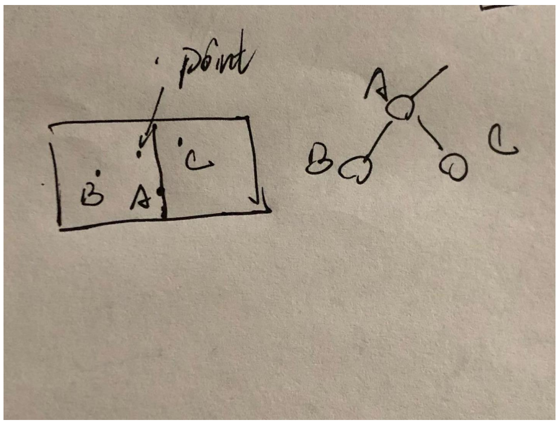


Figure 5: sub tree example

Therefore, we need to check some other sub trees. But we can use some tricks to reduce the time of search.
By checking the node along the path as well as the sub trees, we can find the nearest neighbor. However, we need some tricks.

2.2.2 make use of the distance between the point and pivot

By checking the distance between the point and pivot according to the split dimension stored in the node, we can decide whether to check the values to this node and the sub tree of this node. This can largely decrease the search time.
Some part of search function shows how it works

```
unsigned dim = detect->split;
double rectdist = fabs(detect->pivot - point[dim]);
if(rectdist < mindist){}
```

2.2.3 Store data in the leaves

This is in fact a trick of the query process but to change the build method. When split into two parts and assign to the left sub tree and right sub tree, we would not remove the median from the data set and thus all the data would be stored in the leaves. (In fact, the data is stored twice in node and leaves). Although sometimes we need to check twice, we can increase the efficiency in checking the sub tree and ensure the accuracy.

2.2.4 Return K nearest neighbors

We design a priority queue to return K nearest neighbors. Since we need to check many distances. Every time we check the distance, we add the corresponding data to the array. And the array would compare the data input with the data already inside it. Compare with the largest and if the distance is smaller, replace it, Then again sort the array by distance.

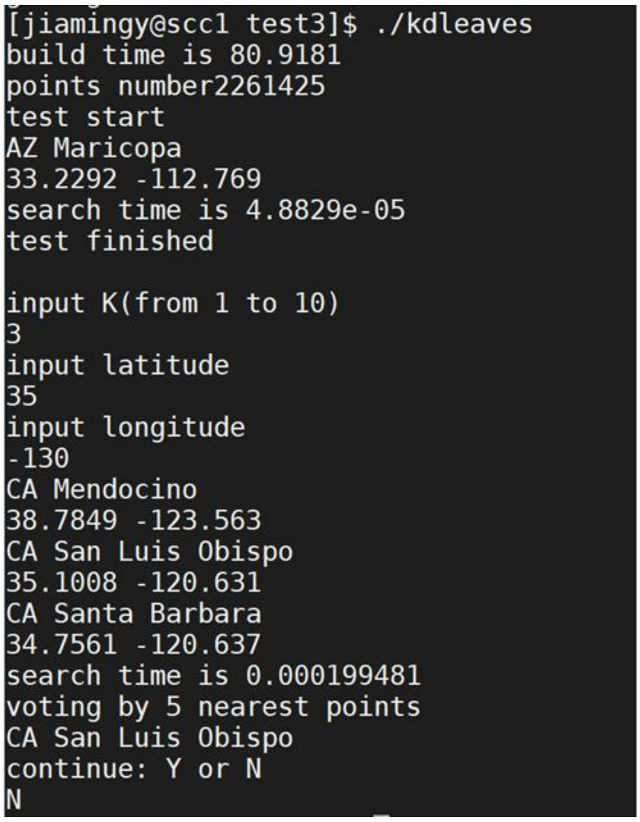
2.2.5 Voting to find state and county

Using KNN above to find 5 nearest neighbors, then use the weighted (assign wights 5,4,3,2,1) voting to return the state and county.

3. Result

Here shows the result of three versions of k-d tree.

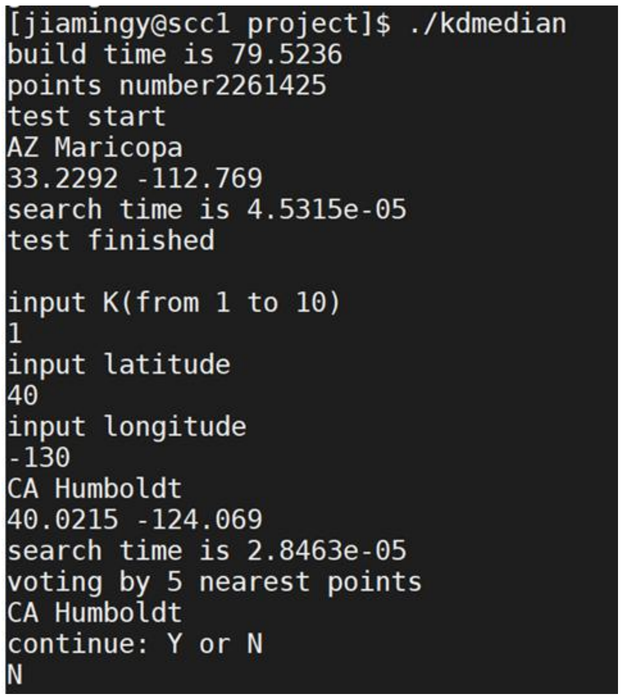
(1)split by median and store data in the leaves.



```
[jiamingy@scc1 test3]$ ./kdleaves
build time is 80.9181
points number2261425
test start
AZ Maricopa
33.2292 -112.769
search time is 4.8829e-05
test finished

input K(from 1 to 10)
3
input latitude
35
input longitude
-130
CA Mendocino
38.7849 -123.563
CA San Luis Obispo
35.1008 -120.631
CA Santa Barbara
34.7561 -120.637
search time is 0.000199481
voting by 5 nearest points
CA San Luis Obispo
continue: Y or N
N
```

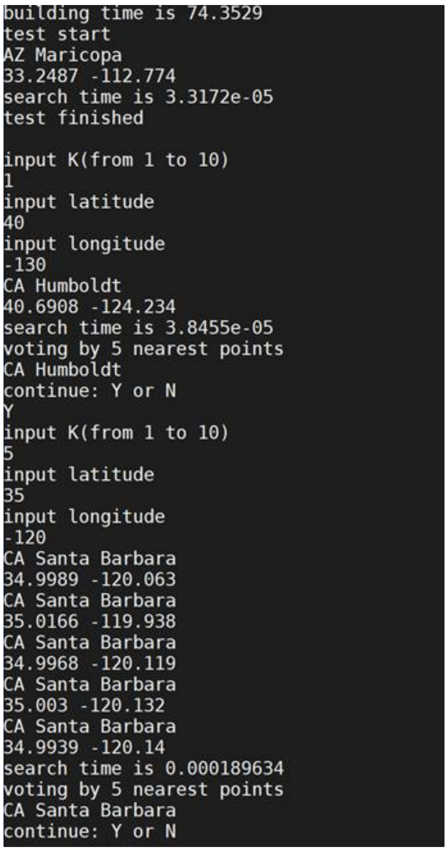
(2) Split by median



```
[jiamingy@scc1 project]$ ./kdmedian
build time is 79.5236
points number2261425
test start
AZ Maricopa
33.2292 -112.769
search time is 4.5315e-05
test finished

input K(from 1 to 10)
1
input latitude
40
input longitude
-130
CA Humboldt
40.0215 -124.069
search time is 2.8463e-05
voting by 5 nearest points
CA Humboldt
continue: Y or N
N
```

(3) Randomly insert



```
building time is 74.3529
test start
AZ Maricopa
33.2487 -112.774
search time is 3.3172e-05
test finished

input K(from 1 to 10)
1
input latitude
40
input longitude
-130
CA Humboldt
40.6908 -124.234
search time is 3.8455e-05
voting by 5 nearest points
CA Humboldt
continue: Y or N
Y
input K(from 1 to 10)
5
input latitude
35
input longitude
-120
CA Santa Barbara
34.9989 -120.063
CA Santa Barbara
35.0166 -119.938
CA Santa Barbara
34.9968 -120.119
CA Santa Barbara
35.003 -120.132
CA Santa Barbara
34.9939 -120.14
search time is 0.000189634
voting by 5 nearest points
CA Santa Barbara
continue: Y or N
```

As a result, we can see from the above figures or you can run the code on SCC (the specific position of code is showed in the code part after the abstract in this report). All three versions of k-d tree can allow users to query the k nearest neighbors and voting to decide the state and county. The code with the better balance of efficiency and accuracy is the code splitting by median and store data in leaves. Kdleaves.cpp can allow searching about 5e-5 for one nearest neighbor search and the query for k nearest neighbors requires 1e-4 to 2e-4 on SCC.

4. Conclusion

In conclusion, I implement three versions of k-d tree and they can all be used to search k nearest neighbor points and find the state and county by voting. Although randomly insert can perform better in some points like the suggested search point (33.24, -112.75) but have a poor average performance. The trick of checking the distance between point and split pivot can largely improve the efficiency and the trick of storing the data in leaves can ensure the accuracy and improve the efficiency a bit.

For the future work, some filters for the data set can be considered, and it is also interesting that the problem can be considered as 3-D problem using 3-D tree. And it may also be interesting that choosing the split dimension with the largest variance.

Reference

[1] Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". *Communications of the ACM*. **18** (9): 509–517. doi:10.1145/361002.361007.