Principles of Urban Informatics
Assignment 11

Posted on: 12/01/2014
Due Date: 12/08/2014

Introduction

In this assignment we will explore different uses of an important spatial data structure, namely KD-tree, for data analysis tasks. As a running example we are going to be using a sample of the NYC taxi data set for May 2013. This data set contains pickup and drop-off times and locations along with additional information about the trips. In case you are interested, the entire data set can be found at `http://www.andresmh.com/nyctaxitrips/` but this is not required for the assignment.

We are going to use the scipy KD-tree implementation `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.KDTree.html`. It is likely that scipy is already installed in your python distribution, if this is not the case you can follow the instructions for installation at `http://docs.scipy.org/doc/numpy-1.9.1/numpy-user-1.9.1.pdf`.

The materials for this assignment can be downloaded at `http://goo.gl/J6oGJm`.

Scenario Description

We want to identify hot spots of taxi activity in Manhattan. In order to do so, we want to aggregate the taxi activity across space. If we do the aggregation based on large regions such as neighborhoods or zip codes, we will loose most of the fine spatial details, so we want do do the aggregation in a larger resolution, similar to `http://goo.gl/D9wMmX`. We are given the coordinates of the road network vertices in Manhattan in the file *manhattan_intersections.txt*. Figure 1 shows a map with these intersections.

Problem 1 - Busiest Intersection

In order to visualize the taxi hot spots, we decided to use the vertices shown in Fig. 1 for the aggregation. We are going to associate the pickup locations of each trip in the data set with the closest intersection (based on the Euclidean distance). You are given a sample script *skeleton_problem1.py* that receives the road network vertices file and the trips file as command line parameters (in this order), your task is to:

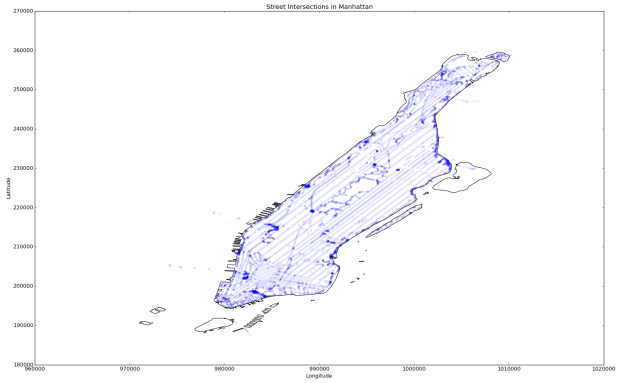

Figure 1: Road network vertices.

a) (5 points) Implement the function *naiveApproach* that receives both the road network nodes coordinates and the trip's pick-up locations and counts for each network node the number of taxi pickups that has that node as the closest one. You should implement this using a naive approach, i.e., for each point loop through the intersections and find the closest one.

b) (10 points) Implement the function *kdtreeApproach* that does the same job as the one in item a), but uses a KD-tree to index (store) the road intersections and to compute the closest intersection for each trip pickup.

c) (Extra Credit: 10 points) In the sample code provided, the execution time is given as an output. Compare the execution times of items a) and b) and report what you observe in a file called *problem1.txt*. Implement the function *plotResults* to plot the counts obtained on a map similar to the Fig. 1. You don't need to include the borough boundary.

Note: For the purpose of this assignment we use the simple approach of considering latitude and longitude as planar coordinates. This is not correct because of effects created by the Earth curvature. The correct approach would be to use the proper distance on the Earth surface or project the points before measuring the distance using the Euclidean norm.

Problem 2 - Busiest Intersection Improved

In the previous problem we assigned each trip to the closest road network node. Although this is a valid approach it suffers from problems, for example, a trip that almost half way of two intersections will be arbitrarily assigned to the closest one although it is almost as close to the second one as to the first one. Also, GPS sensors suffer from noise problems which might change the actual location of the point and therefore change it closest node. A better approach is to count the number of trips within a certain

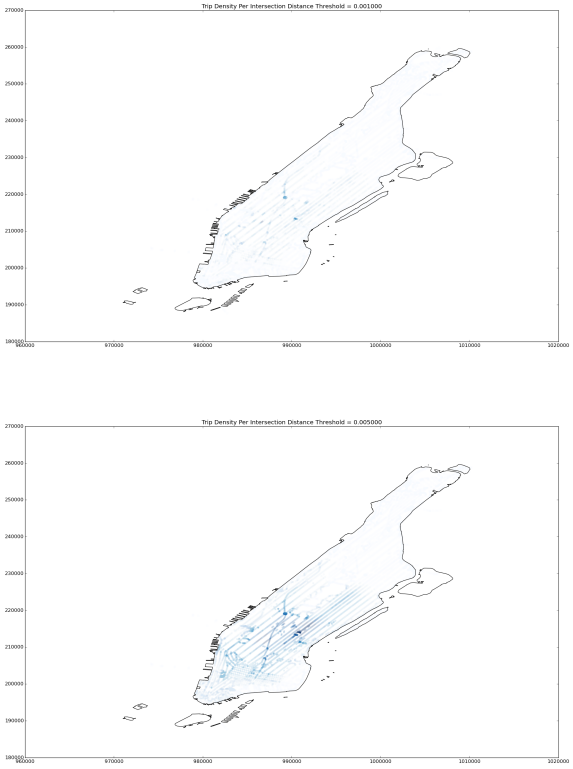

Figure 2: Number of pickups within a radius of each intersection for the thresholds of 0.001 and 0.005 measured in latitude and longitude coordinates.

distance from each intersection. You are given a sample script *skeleton_problem2.py* that receives the road network vertices file, the trips file and the distance threshold as command line parameters (in this order), your task is to:

a) (5 points) Implement the function *naiveApproach* that receives both the road network nodes coordinates and the trip's pick-up locations and counts for each network node the number of taxi pickups that are within a radius of value equal to the distance threshold from the node. You should implement this using a naive approach, i.e., for each node loop through all the trip pickup points and count how many are within the given distance threshold from the node.

b) (10 points) Implement the function *kdtreeApproach* that does the same job as the one in item a), but uses a KD-tree to index (store) the road network nodes and to compute the ones that are within the distance threshold for each trip pickup.

c) (Extra Credit: 10 points) In the sample code provided, the execution time is given as an output. Compare

the execution times of items a) and b) and report what you observe in a file called *problem2.txt*. Implement the function *plotResults* to plot the counts obtained on a map similar to the Fig. 2. You don't need to include the borough boundary.

Problem 3 - Retrieving trips starting from S to E

Spatial data which involve start and end locations without the intermediate path is commonly called origin and destination data. A common task that needs to be performed in this kind of data is to retrieve all the start and end locations pairs (in our example taxi trips) which start in a certain region S and end in a given region E. You are given a sample script *skeleton_problem3.py* that receives the trips file as a command line parameter your task is to:

a) (5 points) Implement the function *naiveApproach* that receives the collection of trip's pick-up and drop-off locations and two rectangles representing the start and end regions respectively and outputs the list of indices (in the input list) of the trips that start inside the rectangular region S and end inside the rectangular region E. You should implement it using a naive approach, i.e., loop through all the trips and test if each of them are inside the given regions.

b)(15 points) Implement the function *kdtreeApproach* that does the same job as the one in item a), but uses a KD-tree to index (store) the trip locations and to query the points that are inside the rectangular regions. In the sample code provided, the execution time is given as an output. Compare the execution times of items a) and b) and report what you observe in a file called *problem3.txt*.

Extra Credit

a) (10 points) Implement a Kernel Density Estimation (KDE) based visualization to improve the visualizations in item c) of problems 1 and 2. You can use as a guideline the visualization in `http://goo.gl/D9wMmX`, but yours does not need to replicate it entirely. You should implement the function *extraCredit* in *skeleton_problem2.py*.

b) (10 points) Generalize the query in problem 3 item b to handle not only rectangles, but also handle polygonal regions. Your function should receive the vertices of the S and E regions and output the indices of the trips that start in S and end in E. You should implement the function *extraCredit* in *skeleton_problem3.py*.

How to submit your assignment?

Your assignment should be submitted using the NYU Classes system. You should submit the sample code files containing your code and also the text files reporting your findings in the performance in each problem. The files should be included in a zip file named NetID_assignment_11.zip, where you should change *NetID* by your NYU Net Id.

Grading

Your programs should be executable. Try to test your code before submitting: your script should solve the problems as requested. The grade is going to be done by testing the correctness of your code with a set of examples. It is always useful to try to visualize the results. Finally, the computations may take some time, so it is always a good idea to develop using small inputs as test cases.

References

- *Scipy KD-Tree*: `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.KDTree.html`