

COL106: Data Structures, I Semester 2016-17

Assignment 7

A taxi aggregator service

October 22, 2016

Nowadays services like Ola and Uber have become very popular. These services aggregate the services of a group of drivers who offer their cars as taxis through the platform provided by the company. One feature that makes these services particularly attractive is that their apps inform you of the time it will take for the nearest taxi to reach your current location. In this assignment we will investigate the software structure required to support this kind of service, with a special emphasis, of course, on the data structures aspect.

Mathematical formulation of the problem

Let us assume the road network is modeled by a graph $G = (V, E)$. Each traffic intersection is modeled by a vertex $v \in V$ and the edges, i.e. elements of E , represent a stretch of road joining two traffic intersections. We are also given a cost function $c : E \rightarrow \{1, 2, 3, \dots\}$ which captures the time taken to travel along an edge. For example if we have a node u which corresponds to the IIT flyover crossing and a node v that corresponds to the IIT hostel crossing, then the edge (u, v) represents the stretch of Outer Ring Road that joins these two intersections and $c(u, v)$ will give us the number of minutes it takes to travel on this road. We will assume for simplicity that this time taken is the same in both directions.

A *path* is a sequence of edges $(u_1, u_2), (u_2, u_3), \dots, (u_k, u_{k+1})$, where the destination of one edge is the source of the next edge. Given a path p of edges $\{e_1, e_2, \dots, e_n\}$, we define the total cost of the path p , $costPath(p) =$

$\sum_{i=1}^n c(e_i)$. Assume that there are multiple paths to go from vertex u to v , namely $\{p_1, p_2, \dots, p_k\}$. Then we define

$$distanceGraph(u, v) = \min_{1 \leq i \leq k} costPath(p_i),$$

i.e., the distance between u and v is the time taken to traverse the path with the minimum *costPath*. A *shortest path* between u and v is any path (it may not be unique) which achieves this minimum value. We only consider paths which do not have any loop.

We also have a set of taxis, S . For the assignment, we will assume that each of these taxis has a location which is one of the vertices of V , i.e., we begin by assuming that each taxi is available and is located at some intersection. Formally, we are given a function $l : S \rightarrow V$ defining the locations of the taxis.

Assignment 7 [100 Marks]

Deadline: 11:55 PM, 10 November 2016

In Assignment 7, you will get customer requests to go from a vertex *source* to a vertex *destination* and you will have to return the closest *available* taxi. A taxi is termed as *busy* or *available*, depending on whether it is currently servicing a customer request or not. Initially, all the taxis will be marked as *available*. They wait at different vertices of the graph.

We will assume that there is a universal clock that moves in discrete time steps. The time starts from 0 and increases in steps of 1. Assume that a customer requests a taxi to go from vertex *source* to vertex *destination*. Among all the *available* taxis, we will choose the nearest taxi to service the customer request. This taxi will be marked as *busy* from *current-time* till *current-time* + $distanceGraph(taxiPosition, source) + distanceGraph(source, destination)$, i.e., the taxi will be busy for the time it takes for it to reach the customer from its current position, and then again for the time it takes for it to transport the customer from *source* to *destination*. After servicing the customer request, the cab waits at the *destination* vertex for subsequent customer requests. For this assignment, we will assume that an available taxi remains stationary and just waits for a customer request.

Unlike previous assignments, we will not provide a function level guideline for this assignment. You are free to use any Java library or your previously written code. *Please mention clearly during the demo that you have used*

code that you yourself wrote for a previous assignment, or that you have used a Java library. Copying another student's code or copying from internet is still prohibited. As a helping hand, you can refer to the implementation of a similar problem on this page:

http://www.vogella.com/tutorials/JavaAlgorithmsDijkstra/article.html#shortestpath_problem.

Also note that while graph and shortest path-related topics will be taught in class, you are expected to be able to implement this assignment without help from lecture material by looking up sources like the page above.

Here is a one possible approach to implement the assignment:

1. Design a Graph. Given a list of edges, you should be able to create a graph. Implement a debug function to print the graph.
2. Implement the shortest path algorithm. Given any two vertices *source* and *destination*, you should be able to find *distanceGraph(source, destination)*.
3. Introduce the concept of a taxi. The property of taxi should be a tuple (*currentposition, availability*). You can use a set to store all the taxis.
4. Implement the function for servicing a customer without introducing the concept of time. This can be done in two simple steps:
 - (a) Determine the distance from all available taxi's to the customer's *source* vertex. Choose the taxi with the minimum distance. In case there are multiple taxi's at the same distance from the customer's position, choose one randomly.
 - (b) Determine the path to drive the taxi from its current position to customer's *source* vertex, and then to customer's *destination* vertex.

After servicing the customer, the taxi's position should be the customer's *destination* vertex.

5. Introduce the concept of time. This needs just one change. For each taxi, instead of using the boolean flag, you can use an integer to denote the number of time units after which the taxi will become *available*. Design question: how to decrement the time unit denoting each occupied taxi's availability?

The grading for this assignment is on a scale of 100. We have two grade components: coding(70) and understanding(30). Steps 1,2,3,4 in the given approach will count for 70% of the coding component marks. The step 5 will count for 30% of the coding component marks.

The input to the assignment is an *actions* file. The possible messages of this file are:

- **edge src dst t:** This will add a new edge to the graph. *src* and *dst* are strings which represent two vertices of the graph. If any of this vertex is not present in the graph, add the node to set of graph vertices. *t* is an integer which represents the time taken to traverse the edge (*src*, *dst*). Assume that the name of any vertex or a taxi contains no spaces or special symbol.
- **taxi taxiName taxiPosition:** This will add a new taxi to the graph. The name of the taxi is denoted by the string *taxiName* and *taxiPosition* is the name of a vertex in the graph. If there is no vertex in the graph with name *taxiPos*, print an error and exit.
- **customer src dst t:** This denotes a customer request to go from the vertex *src* to *dst* at time *t*. Your program should generate the following output:

Available taxis:

Path of taxiA: A, B, C, source. time taken is tA units

Path of taxiB: P, Q, source. time taken is tB units

Path of taxiC: S, Q, C, source. time taken is tC units

Path of taxiD: X, Y, Z, A, B, C, source. time taken is tD units

** Chose taxiB to service the customer request **

Path of customer: source, P, X, Y, destination. time taken is tF units

- **printTaxiPosition t:** This should print the position of the taxi's which are available at time *t*. Your program should generate the following output:

taxiA: A
taxiB: P
taxiC: S
taxiD: X

If you do not follow the expected output format, we will **deduct** 50% of your coding marks.

The assignment supporting files contain two maps: `map1.txt` and `map2.txt`. A representative image for these files are available in `map1.pdf` and `map2.pdf` respectively. If there is a minor mistake in spelling any of the entry in any supporting file, kindly correct it yourself. In order to use `map1.txt`, rename it as `actions.txt`, and execute the program(`java checker`). Test your assignment with `map1.txt` before testing it with `map2.txt`.

Extra Credit [40 marks]

Deadline: **11:55 PM, 10 November 2016**

This is an extra credit part. Here, we have made the taxi search problem slightly more realistic. Firstly, the taxis will always be in motion. Either they are wandering through the graph or they are servicing a customer request. Just like above problem statement, the taxis are located at different vertices at the start of the program. However, instead of waiting at the vertices of the graph, each taxi chooses one destination vertex and starts moving towards it. Once it reaches the destination, the taxi chooses another destination and starts moving towards it. A taxi changes this path only when it is chosen to service a customer request. For the sake of simplicity, let us assume that a taxi can immediately take a U-turn from any point on the map. When a customer requests for a taxi, search for a taxi which can reach the customer at the earliest; irrespective of whether the taxi is currently available or it is servicing a customer request.

The position on the map is defined in a more realistic form; it is now a triplet $(vertexA, vertexB, t)$. This denotes a position on the edge $(vertexA, vertexB)$ which is t time units away from the $vertexA$. Assume that the weight of the edge $(vertexA, vertexB)$ is 4 time units. Going from vertex A to vertex B, we can define five positions on this edge starting from $(vertexA, vertexB, 0)$ i.e. the $vertexA$ to $(vertexA, vertexB, 4)$ i.e. $vertexB$. Note

that the same point can be defined in multiple ways: vertexA is (vertexA, vertexB, 0) and (vertexB, vertexA, 4). So, use your own scheme to define a unique identification for the positions on the map.

As discussed earlier, an available taxi is always moving; it chooses a destination vertex and moves towards it. Let us understand the procedure to choose this destination vertex.

- Each vertex in the graph has a unique name associated with it. Sort all the vertices on the basis of the alphabetical order of their names (https://en.wikipedia.org/wiki/Alphabetical_order). We define the identifier of each node as its position in this sorted list. Assuming that the graph has n vertices, the value of the identifier ranges from 0 to $n - 1$.
- For a taxi T , we define its *nearest - vertex* as the vertex which is located at a minimal distance from its current position. If a taxi is at vertexA, vertexA is its nearest vertex. If the taxi is located between two vertices A and B , the nearest vertex is the vertex at a minimum distance from the current position of the taxi. If both vertices are at an equal distance, the vertex with a smaller identifier is the nearest-vertex.
- Assume that a taxi's nearest-vertex's ID is x . Then its new-destination is computed as

$$new - destination = ((x + 1)) \% (number\ of\ nodes\ in\ the\ graph).$$

Kindly note a few points:

- Assume that the action messages in the actions.txt file are sorted in the increasing order of time.
- We have modified the action message for the customer request query. In this part, it will be specified as
`customer a1 b1 t1 a2 b2 t2 t`
This specifies a customer request to go from $(a1, b1, t1)$ to $(a2, b2, t2)$ at time t . While printing the path of a taxi, only include the vertices of the graph. Do not print the intermediate points on the edges.
- When an available taxi chooses a new destination vertex, print the decision "At time t , *TaxiA* chose a new destination vertex *VertexB*".

The assignment supporting files contain single map: `actions.txt`. A representative image for the file is available in `map1.pdf`.