**AUTONOMOUS VEHICLE ROUTING CODING CHALLENGE**

**By Sweta Mankala**

***Problem Statement:***

Ride-hailing companies like Uber and Lyft have become ubiquitous means to get around cities. Efficient vehicle routing, or figuring out the fastest and most efficient way to direct vehicles to pick up and drop off multiple passengers at a time, has become a crucial component of their success.

In the near future you are the proud owner of an autonomous vehicle! You want to let some friends use it whenever they want so you make an app to let them call the vehicle to come pick them up. In this problem you will develop a system to keep track of the state of your car and your friends.

Here are some more constraints to help you code your solution:

The city is a grid of perpendicular blocks. At initialization your code should take as inputs the number of streets in both the x and y directions (ex. x=10, y=10).

The car can move in any direction exactly 1 block at a time. For our purposes, we only care about when the car is at a vertex of two intersecting streets, so continuing our example, a car can be at the intersection (0, 0) in the upper-left corner of our city, or (9, 9) in the bottom-right corner.

There should be a function to advance time by one time-step (and hence the vehicle by one block in any direction). This function should accept a list of ride requests from your friends. These ride requests come in the form of a list of JSON elements like so:

{ requests: [  
{  
 'name': 'Elon',  
 'start': (3,5),  
 'end': (8,7)

},

{  
'name': 'George',  
'start': (1,2),  
'end': (4,3)

}  
]}

At each time step this list of requests can be an empty list (ex. []), or infinitely long. At each time step you should print out the current position of the vehicle and the names of all of the passengers currently in the vehicle, as well as anybody being dropped off or being picked up at this intersection. (We will assume your car can fit infinitely many people at a time.)

***Proposed Solution:***

Let us assume the city is a grid with x and y dimensions that holds a cell for the car to navigate. The cells marked as ‘1’ represent obstacles or buildings in the grid and the cells marked ‘0’ represent path to navigate for the car.

For the car to navigate, it takes in requests from passengers at any given time (adhoc) which has to be handled at the beginning of each time step. Also, each request that contains a pickup and drop location corresponds to a point on the grid. For this reason, we can assume that passenger requests be streamlined and its visibility to the car is made obvious to keep a count on the number of pick ups and drop offs.

For the search algorithm, out of the most popular A\* search, Hybrid A\* etc., I used a simple shortest minimum path algorithm (DFS or BFS) for an understanding and working implementation of the navigation of the car through the grid provided. With the algorithm being sampling based, and the given environment being discrete, we can encounter a lot of possible paths to finally choose the best optimal path to arrive at the destination.

The software designed has the following functionalities –

1. *gridSearch* – This function helps check the validity of the grid cell and calls the shortest-minimum path algorithm for navigation.

2. *getSuccessors* – This function is used to retrieve locations from passengers and store them in a vector to prioritize and feed them to the gridSearch function according to the requests received.

3. *costFunction* – This function makes the calculation of the time steps being taken towards every movement the car makes towards a grid cell only to optimize and understand the maximum time spend in the car by a passenger.

4. *dfs* – This function makes a depth first search across the grid by checking the 4 directions – up, down, left and right to check if there is an obstacle present in the next immediate grid cell to move forward.

5. *passengerDetails* – This function creates the passenger details with their source and destination being appended into the priority queue.

The idea of the solution that is proposed is to make sure that the car travels with every time step recorded (every move is counted) which helps to understand what the cost function is and how we can further optimize it accordingly.

***Approach:***

A node is created for every passenger request with (<Passenger>, <Source location>, <Destination Location>) and store them in a priority queue to retrieve our passenger’s nearest location. The idea of getSuccessors() is to call out the function passengerDetails() that creates the pair with the details and pushes it into the priority queue as declared in the beginning.

The idea of the priority queue is to fetch elements that have the closest or minimum values to the source and in this case, it was set to the minimum values. It can be further improvised with a custom comparator using Euclidean distance between the cab’s current location and the passenger’s source point fetching the nearest point that can help us identify which passenger to pick up first. The first iteration for gridSearch is made to the cab’s location and the passenger’s current location for it to navigate. The second gridSearch iteration is from the passenger’s source to the destination which are fetched from the current passenger details of the priority queue. We also initialize vectors that calculate and store the timeStep for every navigation call made to gridSearch under two viewpoints – time taken for the cab to make the entire tour in picking up the passengers and dropping them back (J1) & maximum time taken for a passenger to pick up and drop off at their locations (J2). Here, we have considered the possibility of pool ride or ride share as we so popularly call it where multiple passengers get picked up to ride in the cab pool and get dropped off in the order of their location points. This puts some time constraints on the entire process which is explained in depth below.

The function gridSearch basically checks for conditions such as if the location is valid, if it is blocked by an obstacle and if the source and destination are the same points. It recursively calls the depth first search to make the navigation. The depth first search function holds a time complexity of O(RxC) where R is the number of rows and C is the number of columns. With the globally declared timestep variable, it increments itself for every move made (left, right, up and down). This search algorithm is called recursively until the destination is reached and hence is by default not optimal but triggers the search for the best possible path available.

The function costFunction namely J1 and J2 identify the maximum or highest ranked element in the vector parsed to calculate the cost for the same. Since timestep is a globally declared variable used in multiple functions, we use this to calculate the cost of the traversal of our car.

|  |  |
| --- | --- |
| A  1 | B  2 |
| D  <Cab> | C |

For example, let us assume the above grid (2x2) is the city and there are two passengers at locations A and B. Passenger 1 wants to get from A -> D and passenger 2 from B -> D. There are two possible ways to approach this.

For the node => [Grid Location, number of passengers picked up, number of passengers dropped];

Approach 1:   
[D, null, null] -> [A, 1, null] -> [D, 0, 1] -> [A, 0, 1] -> [B, 2, 1] -> [C, 2, 1] -> [D, 0, (1,2)]

Approach 2:  
[D, null, null] -> [A, 1, null] -> [B, (1, 2), null] -> [C, (1, 2), null] -> [D, 0, (1,2)]

From the above approached, with every link being counted as a timestep we can assume that approach 2 is more optimal compared to approach 1. Hence, the cost functions try to help us understand which has an optimal route.

Overall, the code is structured to give an idea of a self-driving car accepting pick up locations from customers to optimally drive itself through a city with least time spent per ride and the time taken overall with reduced fuel or energy consumption making it a better product for future investments.

***Constraints:***

Some of the constraints provided in the above structure that could be improved upon are –

1. Prioritizing the passenger requests based on the cab location.  
2. Time taken for the recursive calls to the search algorithm to find the most optimal path to the destination  
3. Cost function to be used as a parameter to measure the distance for rides overall and feed it back to optimize rides encountered in the future.