CPME 365 Uber Project

Cache Angus

20000629

November 16th, 2018

# The Code

The code takes the network of streets given and assigns them to a graph using the network x imported package. This assigns the values to nodes and edges. After importing the graph, the next step is to utilize this graph in finding the shortest total wait time for the passenger requests.

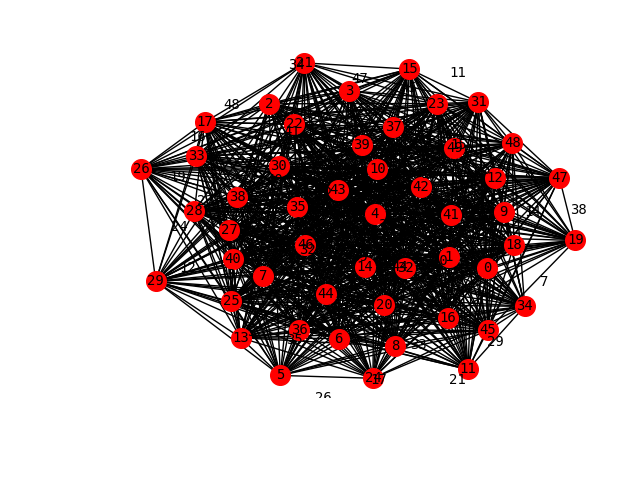


Figure 1: The network as a graph with laels on nodes and edges

To do this I created a function called waitingTime that took the parameters of a graph, a request file name, and a dict of cars. If there were no number of cars given, then the function assigned 2 cars with the given attributes immediately.

The cars need to have three attributes at a minimum to keep track of where they are, how far they are in relation to the next request, and when they are next available. These are all required because each car needs to compare where it is to the other cars and to the request location. This is used in determining which car will be chosen to pick up the request in the least amount of time.

The request file name is sent through a csv read to extract the three different values from each line and put them in their respective array. The values represent the time the request is sent, the location of the request to be picked up, and the final destination of the passenger.

All of the requests need to be filtered through to add each passengers’ waiting time to the time\_waiting value returned at the end of the function, so the primary body of the function is encapsulated in a for loop doing just that. Each car needs to compare how far it is from the passenger to determine the car to pick them up, so another for loop is implemented to check the distance. This check uses Dijkstra’s algorithm on the path lengths in the graph to check all the possible ways to get from one node to another and returned the shortest distance. Each car was also checked to see if they were ‘available’ to pick up the passenger by checking when the request was against the busy time attribute of the car. If the request was after, its value was changed to the new request time.

Each request also needed to be checked against each car and how long their wait time is to finally determine which car would take the request. This was done by adding the available time and the time it took to get to the location for the total elapsed time since all of the requests had begun.

Once the car was chosen it was a simple task of getting the actual passenger wait time, not jus the total elapsed time. If the available time was after the time of the request, the wait time was the total elapsed time minus the request time. This accounts for the additional time it took for the car to finish its latest drop off before making its way to the next passenger. Otherwise the wait time was simply how long it took to get to the request start location.

The car attributes needed to be updated to reflect the lasts passenger’s full trip. This was done by finding how long it would take to complete the actual trip requested, once again using Dijkstra’s algorithm. This was the second attribute of the car array. The next available time for the car is to be the time it takes to get to the passenger and the time it takes to finish the trip. The actual time\_waiting is the variable additional (which is the wait time) being added from each passenger.

This function is implemented in two different ways. Once with the base of 2 cars and once with different amounts of cars for comparison.

# Design Choices

It is evident from the description that the simplest way to solve the uber problem is to use some form of a greedy solution and involve Dijkstra’s Algorithm. The reason that this is evident is because the problem asks for the total time of passengers waiting and wants to minimize this value. Since the cars needs to travel the roads, they need the fastest way in between which ever point they are at to wherever the destination is. They also need the fasted way to get to the next customer. This is the entire premise of Dijkstra’s Algorithm. The rest of the solution is also based on a greedy solution since it is essentially a scheduling problem. Since the start and end times of each passenger need to be compared and the amount of cars needs to be accounted for, this is best solved by greedy “scheduling.”

On the note of scheduling, there were some alterations to a basic scheduling method. These come from the fac that there is no sorting algorithm to deal with the data coming in. As the requests were already in ascending order, there was no need to sort them. Instead of sorting, there was a need to choose between the cars. This is where the more typical scheduling comes in, as the cars are compared to one another and then added or replaced as the chosen car.

Using network x imports was the simplest and only sensible way to graph the csv file in python as a graph.

The reason it was chosen to make the car feature a dict is because this is the simplest way to keep track of multiple attributes for the car and do it for multiple cars at the same time. In python, the user is able to create these lists of lists. Storing the cars in this form also made it easy to iterate through the different cars and to access the different attributes of each list.

I put the opening of the csv for requests within the waitingTime() function because it allows the function to be called with different requests without having to re-write the csv call each time.

# Complexity

The complexity of the problem can be broken down into the different aspects of the code. For the waitingTime() function there is:

* Dijkstra’s Algorithm is O(E + V log(v))
* Iterating through the requests: O(n) where n is the number of requests
* Iterating through the cars O(c) where c is the number of cars

Starting from the inside out, with a Dijkstra’s algorithm inside of a for-loop, that is O((E +Vlog(v))\*c). There is another call to cycle through the number of cars, O(c). These are added and called recursively, making them O(n\*(E+Vlog(v))\*2c). Then the Dijkstra’s algorithm is called within the over-arching for-loop for O((E +Vlog(v))\*n). When adding these, then comparing to find the action Big O notation of the problem, the final big O is the largest time, which is O(n\*(E+Vlog(v))\*2c) or O(n\*(E+Vlog(v))\*c).

All if statements are O(1) and as such are not included in the run time assessment.

The actual run time of this solution is highly dependent on the size of each attribute.

When determining the most important factor in the run time of the program, it is good to look at the parameters called. The network or graph created is essentially a permanent function, as it is not altered or accessed anywhere in the code except for where it is first initialized. Though there may be a lot of nodes—and in turn from that a lot of edges—once the graph is made it is not changing. The amount of cars may change but there will never be more cars than there are pick up locations or edges. This leaves the amount of requests as the most volatile, and therefore the most effective, variable in the run time.

# Testing

To test the code, there were multiple trials run. These trials checked the two different sets of request data, request.csv and supplementalpickups.csv , to ensure multiple things. The first check was the ensure that they ran and then, baring that they did, the second check was to see if they produced results that were ‘correct’ when compared to other people’s answers.

The first set of data was tested by using the request.csv. This produced a result of 618 minutes for the total wait time. This seemed to be the consensus for around the most accurate value. When checking the multiple car tests they produced a set of values for the total minutes. The total time array looked as such: [618.0, 463.0, 409.0, 379.0, 366.0, 331.0, 307.0]

# Performance Assessment

The solution I developed ended up resulting in the total time passengers spent waiting as 618 minutes. This was the result of having 2 cars that started at position 1 on the graph. Since there was only 2 cars, there was a very high chance of one or both of the cars already having a passenger when the next request was sent. This resulted in the seeming high wait time, though this was the optimized solution (such as required by greedy algorithms). It is reasonable to predict that there will be a decrease in wait time when the number of cars goes up in a roughly linear fashion. When looking at the graph produced by the multiCars dict, the shape takes on a slightly shifted 1/e graph. While not linear it is evident that the relationship predicted of the waiting time going down holds.

A close up of a map

Description automatically generated

Figure 2: The minutes waiting vs the amount of cars

Overall performance of the code shows that there is plenty of room for improvement, or at least alteration. While the Dijkstra’s algorithm is essentially irreplaceable for optimizing run time when finding the shortest roots along graph edges, there is a way to change the greedy scheduling system that I used. Since there needs to be a method to determine which car is to pick up the request, branching and bounding could be used to make this decision. By using a branching and bounding algorithm we are able to make the ‘sequence of decisions’ .

The performance of the code also depended on the hardware it was run on and the program I wrote it in. I unfortunately have a terribly slow computer that runs any coding program at approximately half the speed of anyone else. My ability to judge the overall performance is highly skewed.