CMPE365 Final Project:

Uber Routing using Greedy Heuristics and Dijkstra’s Algorithm

Viraj Bangari 10186046

2018-11-17

# Problem statement

The project I decided to solve was called “Uber”. The dataset consists of an adjacency matrix that describes the weights and nodes of a city and list of passenger requests that contain the time of request, start node and end node. The goal of the project is to schedule drivers to pick and drop off each passenger such that the sum of the a passengers request time and actual pick time is at a minimum. The constraint is that drivers can only pick up a passenger if the current time is greater than the request time. This problem can thought of as special case the NP-hard Travelling Salesman Problem, where approach of generating all possible combinations of request orders is . Generating the optimal solution by brute force is infeasible since the request data set consists of points, and . My approach was an greedy approximation algorithm that uses shortest distance heuristics with dijkstra’s algorithm.

# Attempt 1: Shortest distance heuristics with variable number of drivers and committed number of trips

## Algorithm Overview

The greedy approach consists of list of uber drivers, a set of free drivers, and a queue of passengers waiting to be picked up. An uber driver has a list of assigned trips, which can either be labelled as a “pickup” or a “dropoff”. If a driver particular is assigned a trip, the trip request is stored in an internal queue, and the time for it to complete the trip is added to an internal variable called *committedTripLength*. If a driver finishes a trip, its *committedTripLength* is reduced by the time of the trip it just took and the queue is popped. If a driver finishes a “pickup” trip, the differences from the time of pickup to the time when that particular passenger requested that trip is stored in a variable called *totalWaitingTime*. The set of free drivers are the uber drivers that are willing to commit to a pickup a passenger at some point in the future. The willingness of driver is settable by a variable called *maxCommittedTrips*. This is the maximum number of trips a driver can be assigned to at a time. If the variable is 0, a driver can only be assigned to a passenger if it has no committed trips. If the variable is , then a driver will always be willing to commit to a trip. The waiting queue consists of a list of passenger sorted by increasing request time. A passenger is popped from waiting queue if it is assigned to a driver. This is written in pseudocode for the sake of clarity.

graph cityGraph; // graph of the city

queue futureRequests = [p1, p2, .. pn] // Queue of all passenger requests sorted by request time.

queue waitingQueue = [] // queue of waiting passengers

list drivers = [Driver(0), Driver(1), … ] // list of drivers and their starting node

set freeDrivers = {} U drivers; // set of free drivers. Initially it is the list of drivers.

totalWaitingTime = 0

for (int time = 0; time < maxTime; time++) {

if (futurePassengers.empty() && waitingPasssengers.empty() && freeDrivers == drivers) {

// Terminate if there are no future passengers, no waiting passengers and all drivers

// are free.

break;

}

// Add passengers to waiting queue if it is the time of request.

while (futureRequests.front().time == time) {

waitingQueue.push(futureRequests.pop())

}

// If there are free drivers and passengers waiting, try assigning them.

while (!freeDrivers.empty() && !waitingQueue.empty()) {

Passenger p = waitingQueue.pop()

// Get dist uses dijkstra’s algorithm.

distToDropoff = graph.getDist(p.startLocation, p.endLocation);

// Assign driver who will get to the passenger in the shortest time.

minTotalDist = INF

Driver best;

for (Driver d : freeDrivers) {

alt = d.commitedTripLength +

graph.getDist(d.trips.back().endpoint, p.startLocation)

if (alt < minTotalDist) {

minTotalDist = alt;

best = d;

}

}

best.assignPickup(p.startLocation);

best.assignDropoff(p.endLocation);

if (best.trips.size() > maxCommittedTrips) {

freeDrivers.remove(d);

}

}

for (Driver d : drivers) {

// Move all the drivers 1 step in time.

d.step();

if (d.justFinishedPickup()) {

totalWaitingTime += d.timeDiff;

}

}

}

## Implementation of Dijkstra’s Algorithm

An important part of the previous algorithm is the function *getDist.* This is implemented using dijkstra’s algorithm. The following image is my specific implementation of Dijkstra’s algorithm for this project.

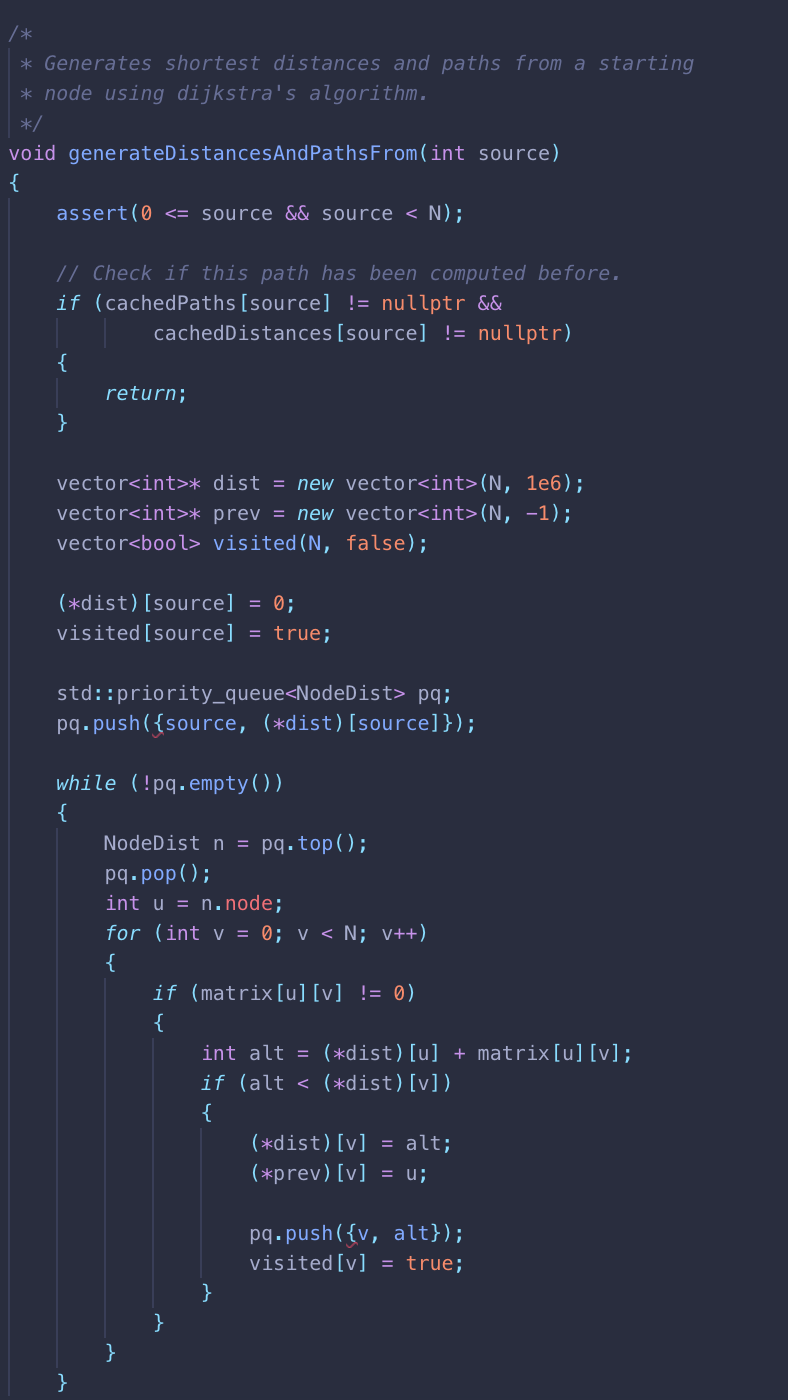


Figure - Implementation of Dijkstra's Algorithm

An important detail in this implantation a full run of Dijkstra’s is not called multiple times from the same source. The computed distances from a starting node are stores in an table of lists, meaning that future calls of Dijkstra’s are retrieved instead of recomputed.

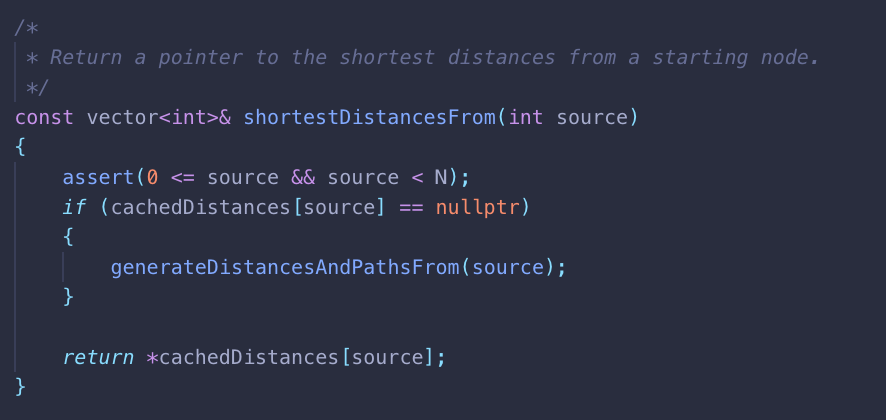


Figure - Cached Dijkstra’s algorithm

## Implementation of Passenger

The data structure for the passenger is quite simple. It stores the request time, start location, end location and an id. The id is assigned by the order that the passenger appears in the parsed text file.

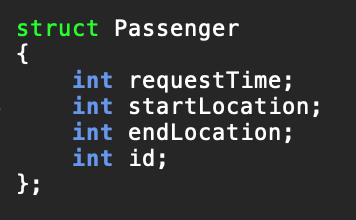


Figure - Passenger struct

## Implementation of Trip

The implementation of the trip data structure is also quite simple. It contains the distance of the trip, the end location, and the associated passenger.

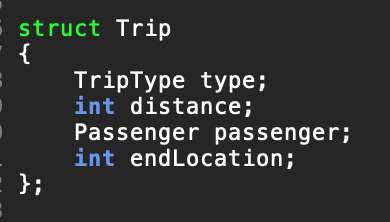
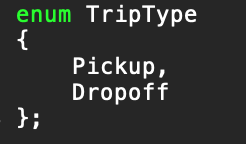
 

Figure - Trip struct

## Implementation of driver and step

The driver is an object that can accept a new passenger, and will create the associated trip for it. An important function is called “step”, which moves the driver one time unit forward. The reason for the step function is because the *maxCommittedTrips* limits how many trips a driver and commit to at a certain time sequence.

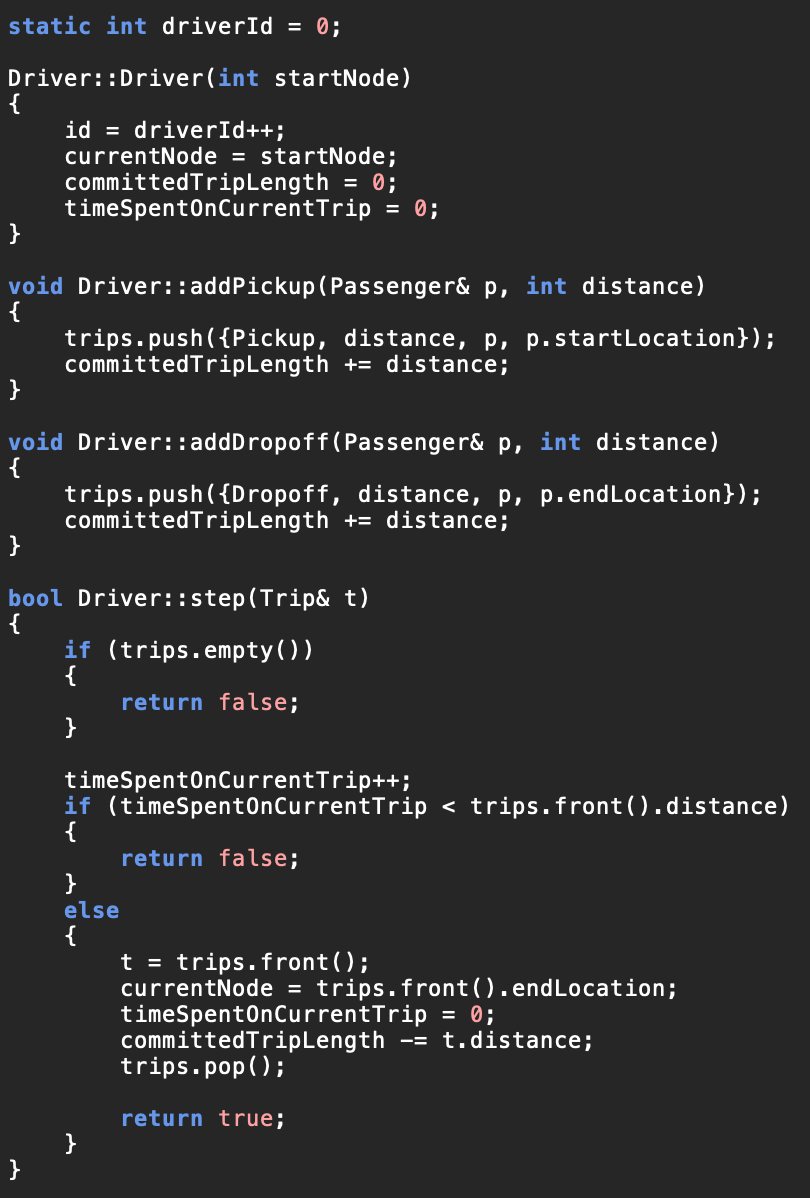


Figure - Implementation of driver

## Runtime complexity

This algorithm uses Dijkstra’s to determine the shortest distance between two nodes. To reduce the overall complexity, one can compute Dijkstra’s algorithm starting from every node in the graph and can store the values in a table. Since I use C++’s min heap, doing this requires a complexity of of . Since the number of vertices in the graph, ~50, and the number of edges, ~2500, are constant and not very large doing this initial computation is worthwhile since it allows finding the shortest distances to have an lookup. Since the size of the graph does not change, this computation takes constant constant time relative to the input data set. Another option would be to use the Floyd-Warshall algorithm which computes the shortest distance between any two nodes in . If the graph had an extremely high number of edges, the Floyd-Warshall implementation would scale better. Considering that

The section the algorithm is the section where passengers are being assigned to driver has an overall complexity of where k is the number of passengers in the waiting queue, and n is the number of free drivers. Technically, this means that the overall complexity of my implementation , where is the maximum time of the simulation.

## Results

Using the initial data set, I ran my algorithm multiple while varying the number of drivers from 1 to 50 and by adjusting the value of *maxCommittedTrips.* The total waiting time was plotted as the number of drivers increased.

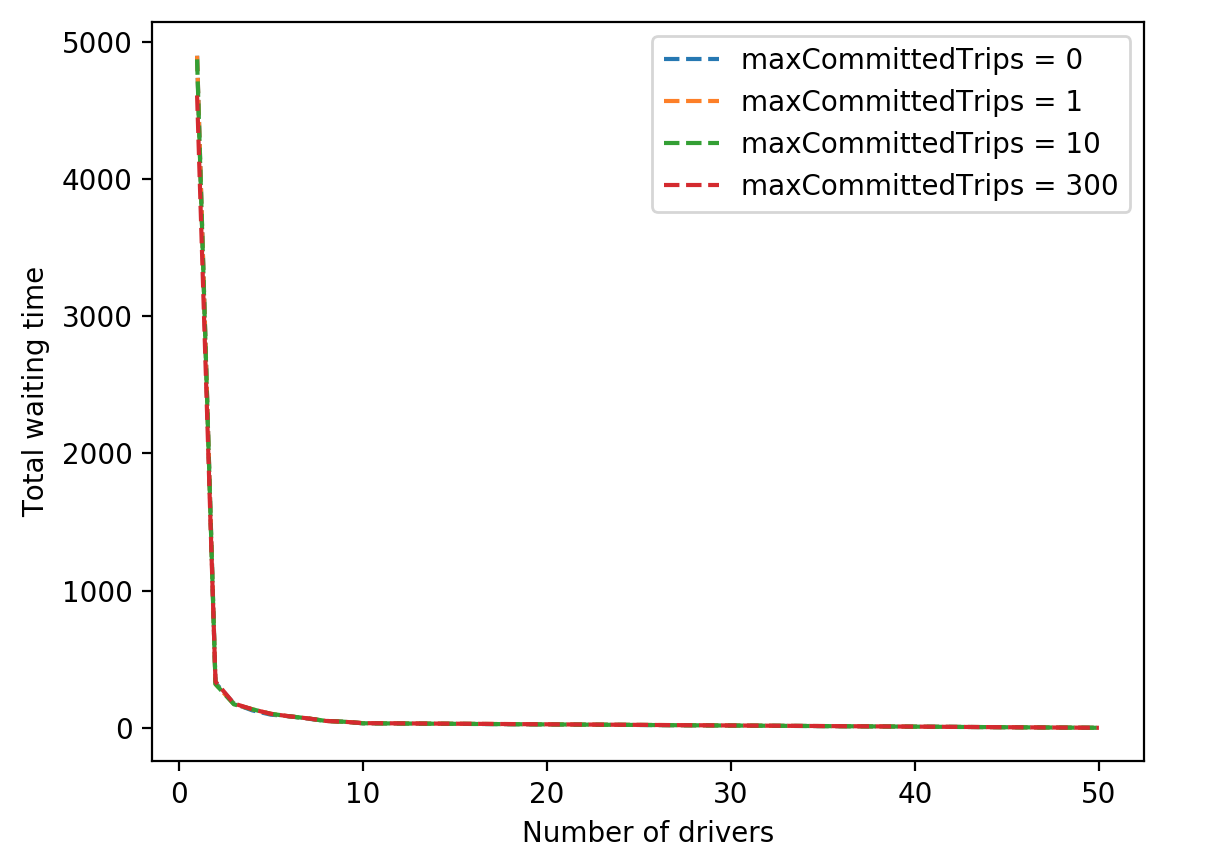


Figure - Total waiting time with varying number of drivers and maxCommittedTrips on original data set

Using the greedy approach, the total waiting time of passengers decayed exponentially as the number of drivers increased linearly. This means that the payoff from adding an extra driver is massive. Another interesting thing to note was that the value of *maxCommittedTrips* did not have a significant effect on the total waiting time as the number of drivers increased. The following graphs not including the next one were plotted with *maxCommittedTrips* set to 0.

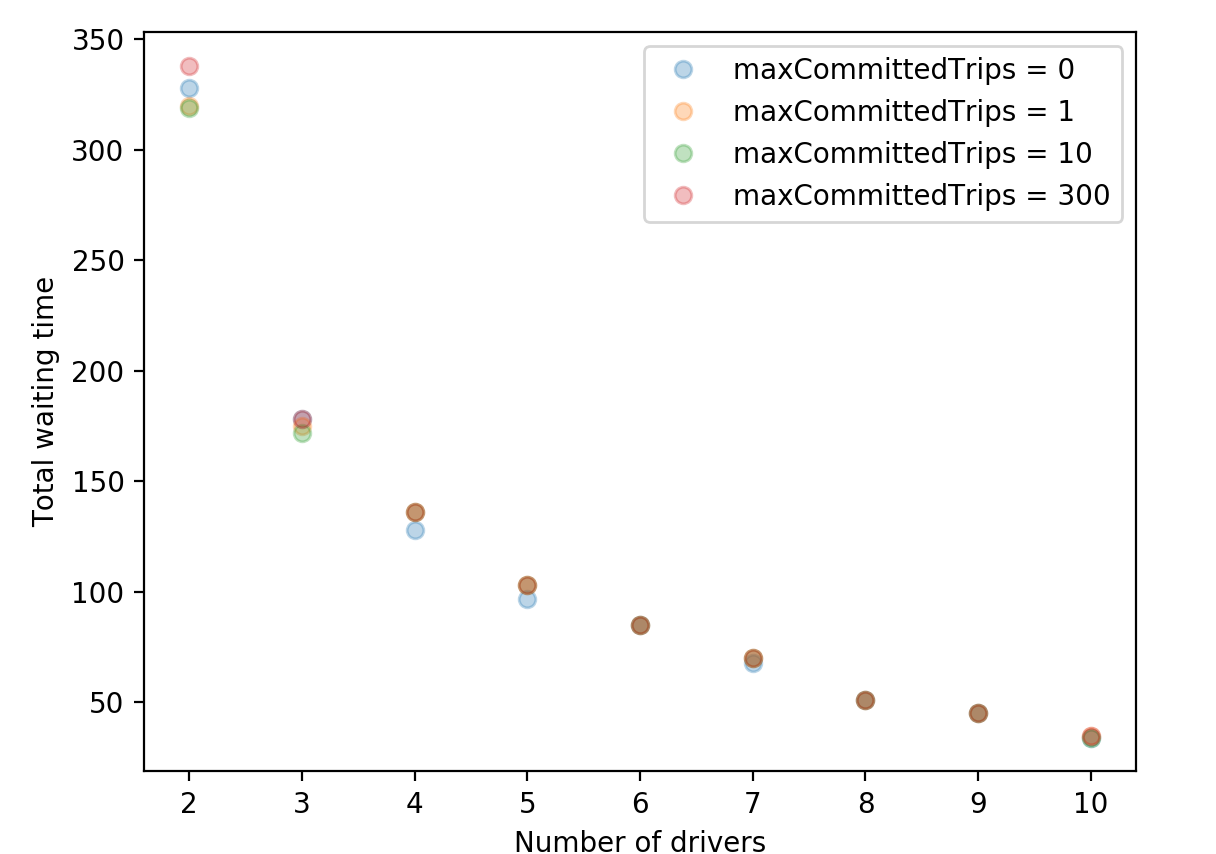


Figure - Effect of maxCommittedTrips on total waiting time on original data set. Some points appear they overlap..

The exponential behaviour was also found when running the algorithm on the supplementary dataset. Though the waiting total waiting time using one driver was significantly larger, the total waiting times approached similar values as the number of drivers increased.

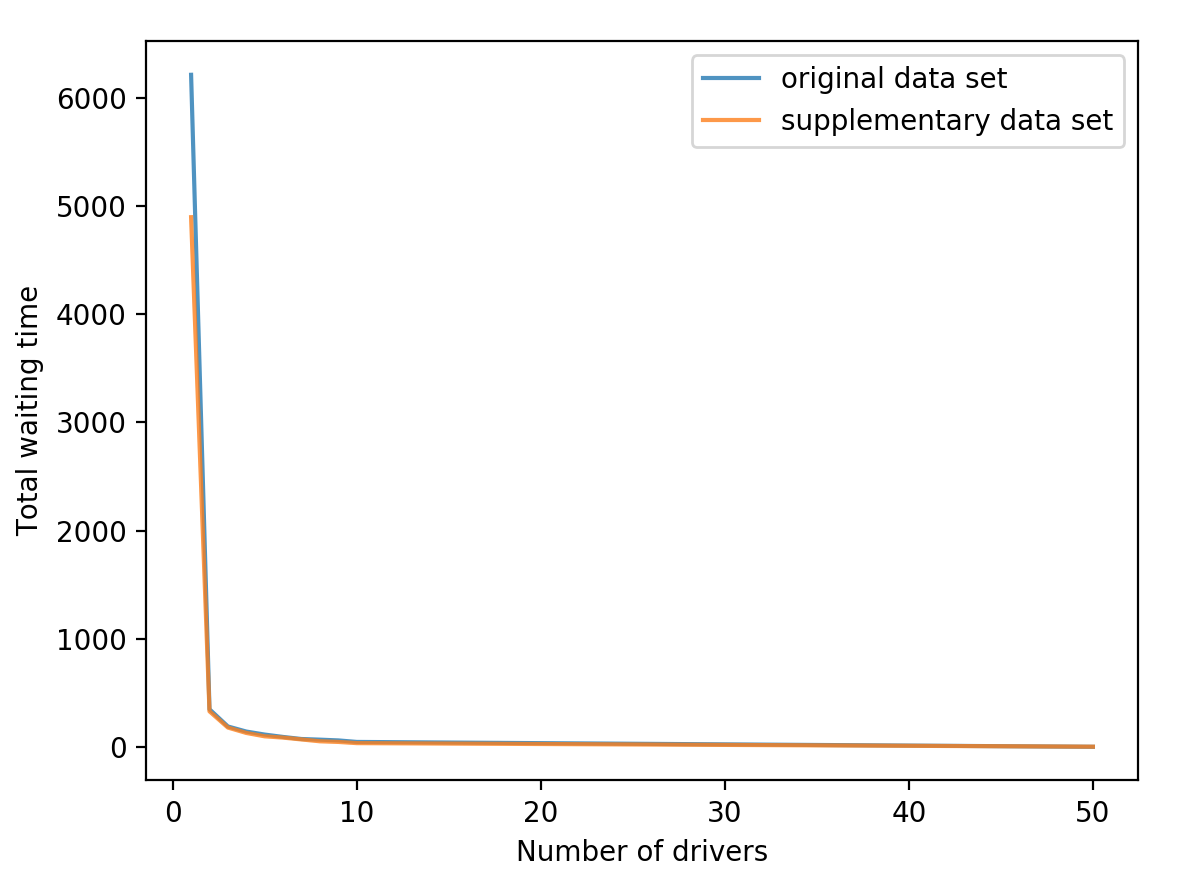
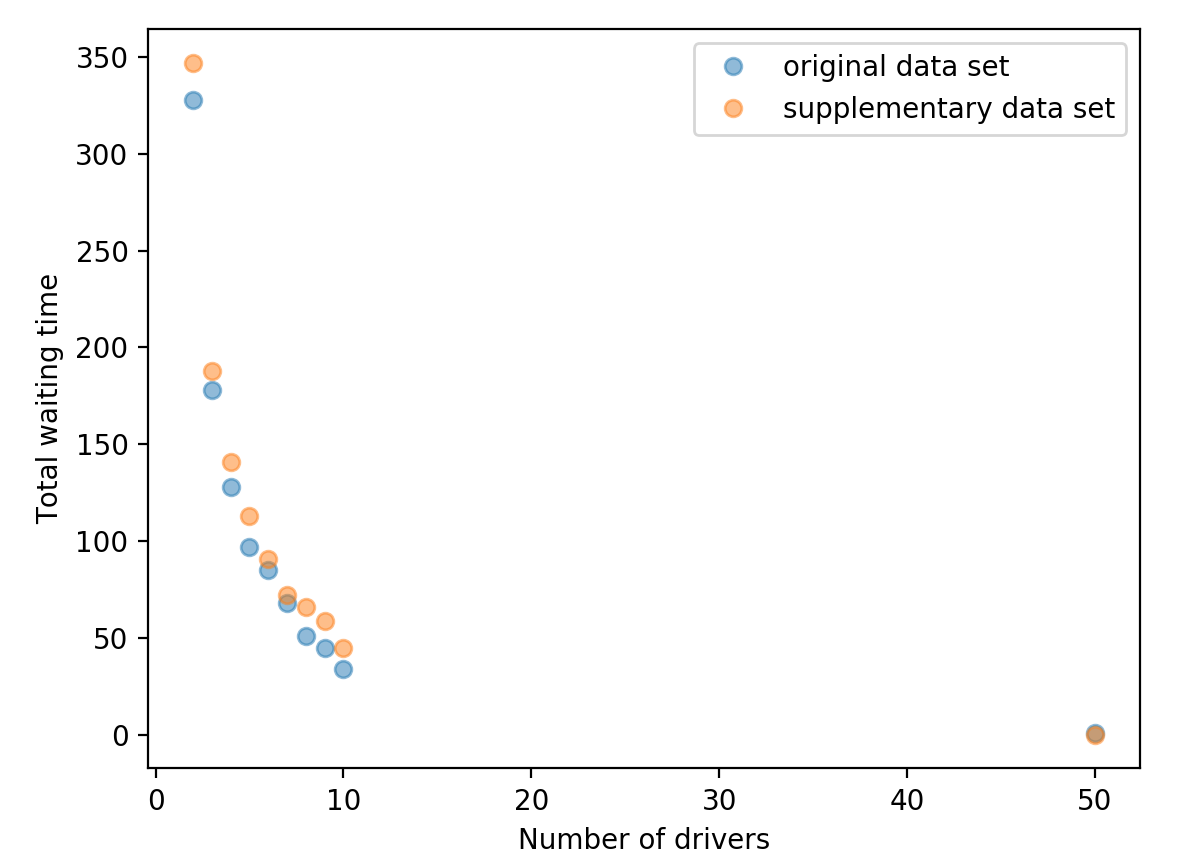
 

Figure - Total waiting time when run on both data sets and increasing number of drivers

A summary of these results are presented in the following table. The elements of the table are the total waiting time and average waiting time when ran with the number of drivers in the column and *maxCommittedTrips* set to 0.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1 driver** | **2 drivers** | **10 drivers** | **50 drivers** |
| **Original Data Set** | 4898 / 16.3 | 338 / 1.12 | 35 / 0.12 | 1 / 0.0 |
| **New Data Set** | 6125 / 20.4 | 347 / 1.15 | 45 / 0.15 | 0.0 / 0.0 |

# Attempt 2: Shortest wait heuristics with variable number of drivers

# Conclusion

Finding the optimal set of combinations for uber dataset in an NP-hard problem. However, a greedy approach using shortest distance heuristics with Dijkstra’s algorithm provided promising results with both data sets. As the number of drivers increased, the total waiting time reduced exponentially. For instance, using only two drivers meant that the average waiting time for a passenger was 1.12 and 1.15 for the first and second data set respectively. Adding fifty drivers drivers meant that the average waiting time was reduced almost 0 time units. This behaviour is expected, as this would be the equivalent of having one driver for each node in the graph. Considering that the algorithm can be implemented in time, this relatively lightweight approach provides a good solution to the problem since generating the brute-force optimal solution has factorial time complexity, and the branch and bound approach has exponential complexity. Considering that the *maxCommittedTrips* parameter did not have a significant effect on the performance of the algorithm, this algorithm was improved to have complexity runtime complexity.