CMPE365 Final Project:

Uber scheduling using greedy heuristics with shortest distance and shortest wait time

Viraj Bangari 10186046

2018-11-17

1 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 n 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 $O(N!)$. Generating the optimal solution by brute force is infeasible since the request data set consists of 300 points, and $300! = 3.06 * 10^{614}$. My initial approach was an greedy approximation algorithm that uses shortest distance heuristics with Dijkstra’s algorithm. My second approach also uses Dijkstra’s algorithm is greedy by the shortest wait time and is pre-emptive.

2 Approach 1: Shortest distance scheduling with variable number of drivers and maximum committed trips

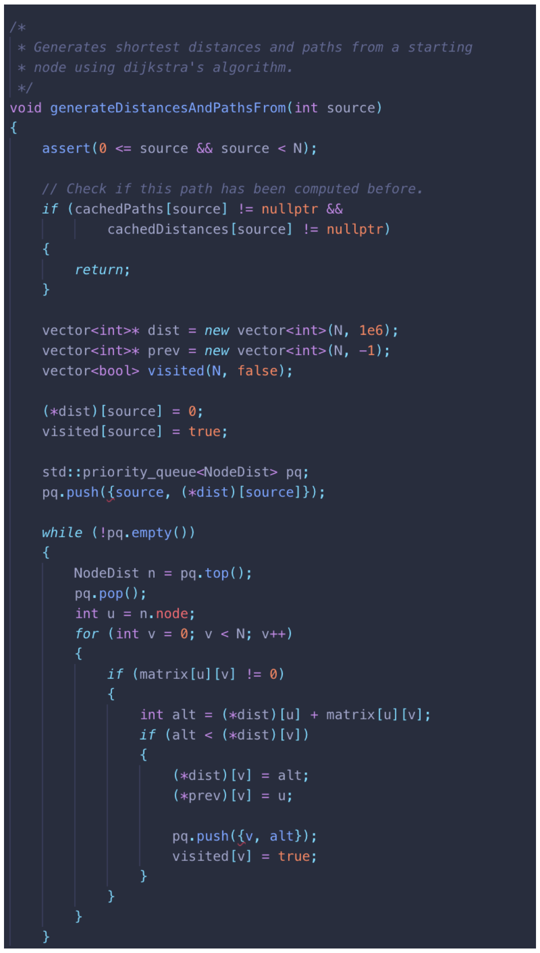
2.1 Algorithm Overview

The greedy approach consists of list of n 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 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. Passengers are assigned to the driver by their shortest distance from the driver's current location, or the final location of their most recently committed trip. This is written in pseudocode below, as the actual implementation as a lot of extraneous parts that take away from the meaning.

```
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;
                }
        }
}
```

2.2 Implementation of Dijkstra’s Algorithm

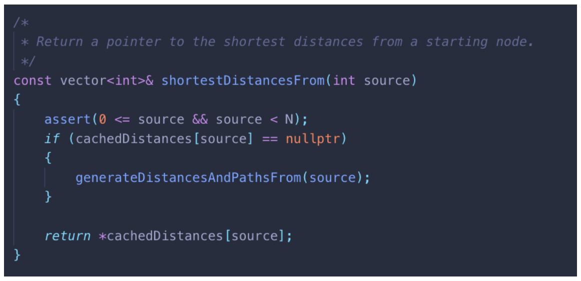
An important part of the previous algorithm is the function *getDist.* This is implemented using Dijkstra’s algorithm. The following image is my specific implementation of Dijkstra’s algorithm for this project.



```
/*
 * Generates shortest distances and paths from a starting
 * node using dijkstra's algorithm.
 */
void generateDistancesAndPathsFrom(int source)
{
    assert(0 <= source && source < N);

    // Check if this path has been computed before.
    if (cachedPaths[source] != nullptr &&
            cachedDistances[source] != nullptr)
    {
        return;
    }

    vector<int>* dist = new vector<int>(N, 1e6);
    vector<int>* prev = new vector<int>(N, -1);
    vector<bool> visited(N, false);

    (*dist)[source] = 0;
    visited[source] = true;

    std::priority_queue<NodeDist> pq;
    pq.push({source, (*dist)[source]});

    while (!pq.empty())
    {
        NodeDist n = pq.top();
        pq.pop();
        int u = n.node;
        for (int v = 0; v < N; v++)
        {
            if (matrix[u][v] != 0)
            {
                int alt = (*dist)[u] + matrix[u][v];
                if (alt < (*dist)[v])
                {
                    (*dist)[v] = alt;
                    (*prev)[v] = u;

                    pq.push({v, alt});
                    visited[v] = true;
                }
            }
        }
    }
```

Figure 1 - 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.

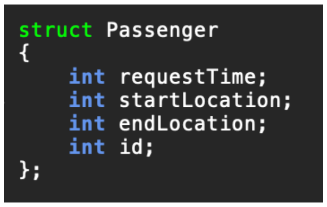


```
/*
 * Return a pointer to the shortest distances from a starting node.
 */
const vector<int>& shortestDistancesFrom(int source)
{
    assert(0 <= source && source < N);
    if (cachedDistances[source] == nullptr)
    {
        generateDistancesAndPathsFrom(source);
    }

    return *cachedDistances[source];
}
```

Figure 2 - Cached Dijkstra's algorithm

2.3 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.

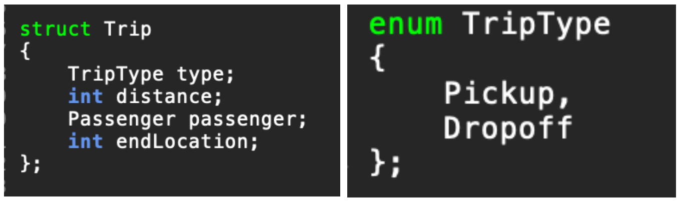


```
struct Passenger
{
    int requestTime;
    int startLocation;
    int endLocation;
    int id;
};
```

Figure 3 - Passenger struct

2.4 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.



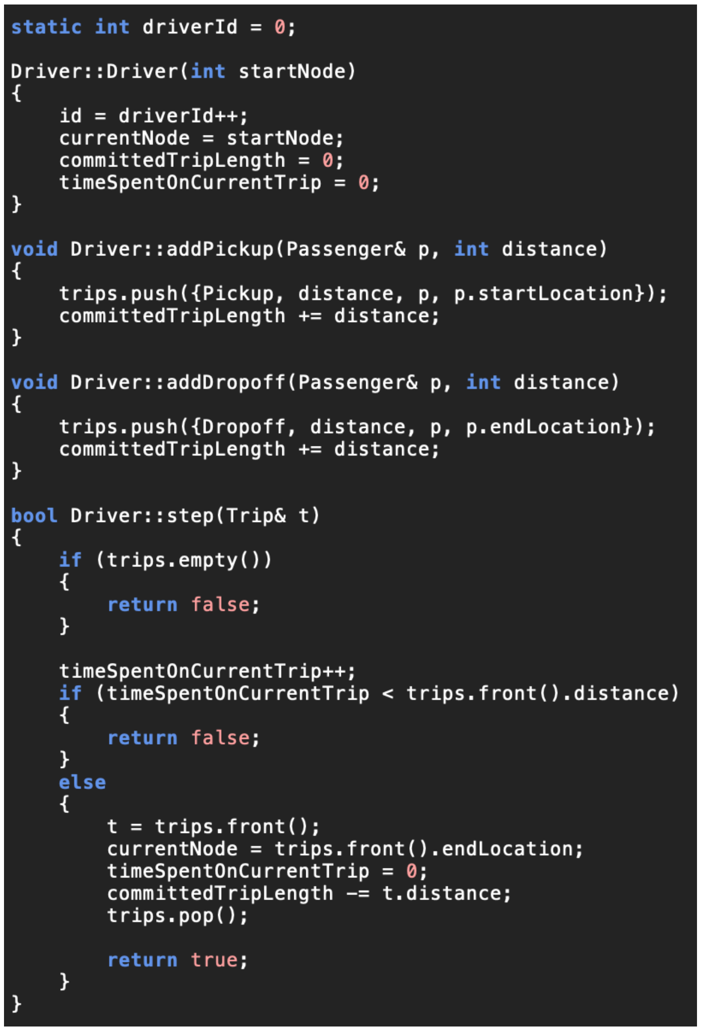
```
struct Trip
{
    TripType type;
    int distance;
    Passenger passenger;
    int endLocation;
};
```

```
enum TripType
{
    Pickup,
    Dropoff
};
```

Figure 4 - Trip struct

2.5 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.



```cpp
static int driverId = 0;

Driver::Driver(int startNode)
{
    id = driverId++;
    currentNode = startNode;
    committedTripLength = 0;
    timeSpentOnCurrentTrip = 0;
}

void Driver::addPickup(Passenger& p, int distance)
{
    trips.push({Pickup, distance, p, p.startLocation});
    committedTripLength += distance;
}

void Driver::addDropoff(Passenger& p, int distance)
{
    trips.push({Dropoff, distance, p, p.endLocation});
    committedTripLength += distance;
}

bool Driver::step(Trip& t)
{
    if (trips.empty())
    {
        return false;
    }

    timeSpentOnCurrentTrip++;
    if (timeSpentOnCurrentTrip < trips.front().distance)
    {
        return false;
    }
    else
    {
        t = trips.front();
        currentNode = trips.front().endLocation;
        timeSpentOnCurrentTrip = 0;
        committedTripLength -= t.distance;
        trips.pop();

        return true;
    }
}
```

Figure 5 - Implementation of driver

2.6 Runtime complexity

This algorithm uses Dijkstra's to determine the shortest distance between a starting node and all other 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 $O(|E||V| + |V|^2 \log(|V|))$. 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 $O(1)$ lookup. Since the size of the graph does not change, this computation takes 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 $O(|V|^3)$. 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 $O(kn)$ where k is the number of passengers in the waiting queue, and n is the number of free drivers. This means that the overall complexity of my implementation $O(tkn)$, where t is the maximum time of the simulation.

2.7 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.* Since there were 300 points in the input, setting *maxCommittedTrips* to 300 resulted in the drivers always being free. The total waiting time was plotted as the number of drivers increased.

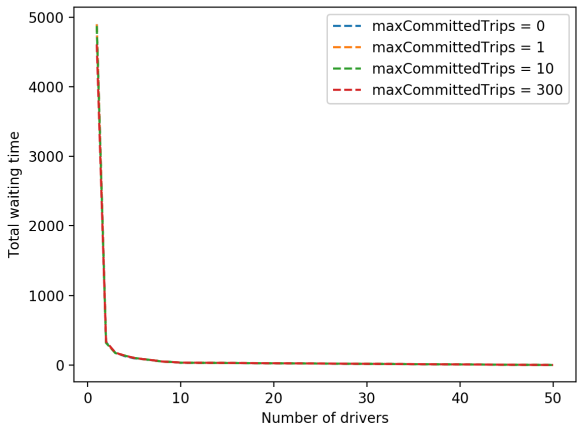


Figure 6 - Total waiting time with varying number of drivers and maxCommittedTrips on original data set

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.

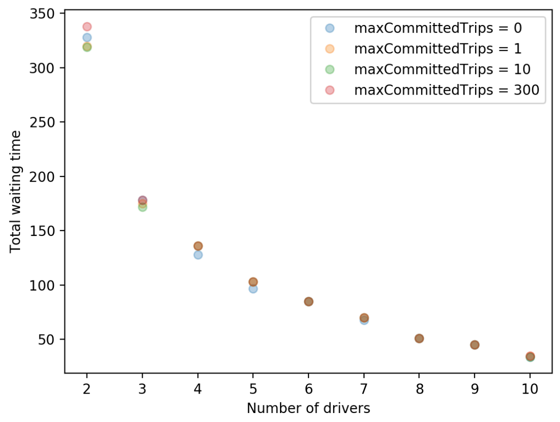


Figure 7 - Effect of maxCommittedTrips on total waiting time on original data set. Some points appear they overlap..

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.

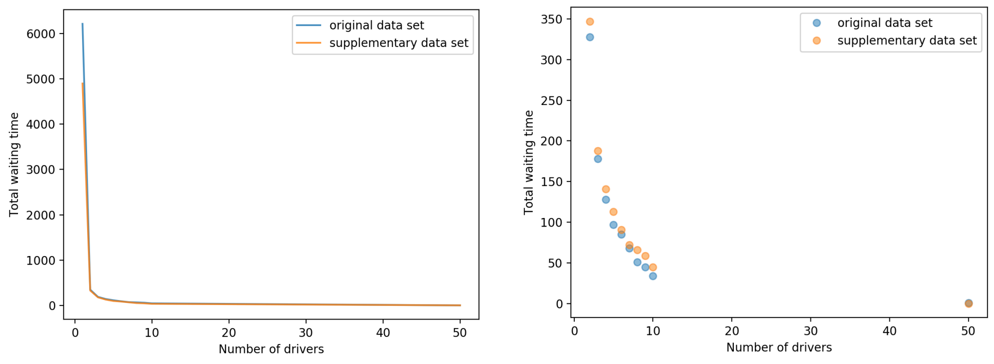


Figure 8 - Total waiting time when run on both data sets and increasing number of drivers

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.

Table 1 - Summary of results from shortest distance approach

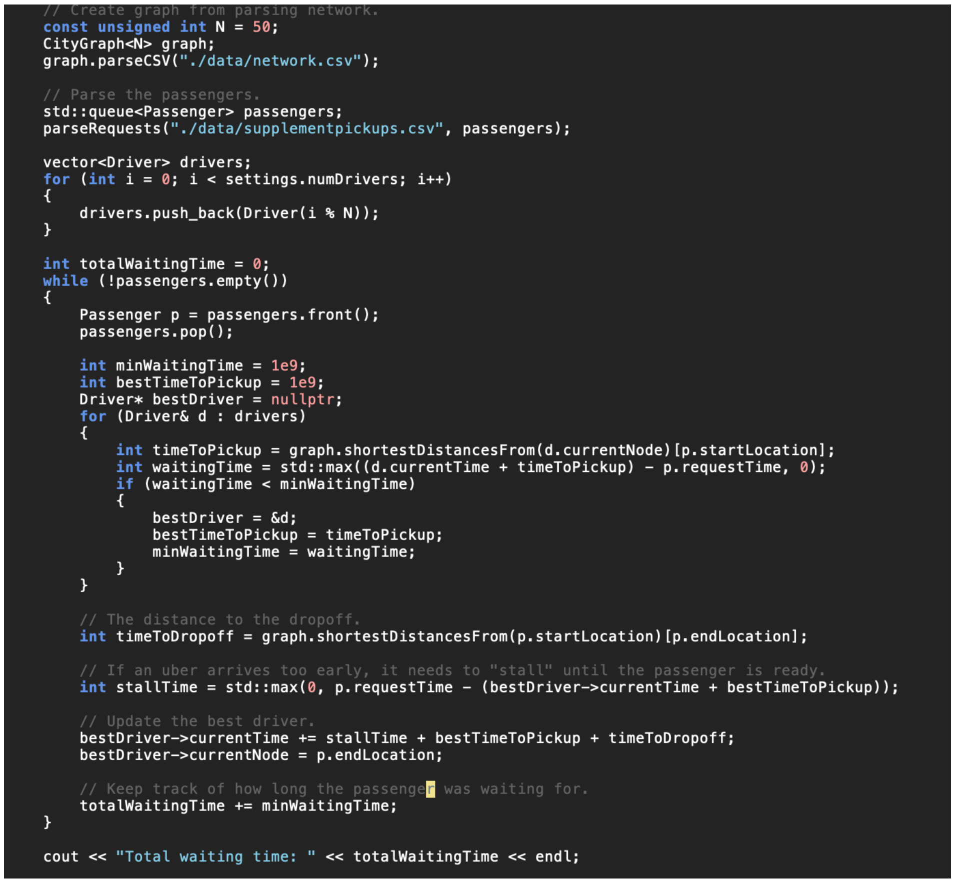
	1 driver	**2 drivers**	**10 drivers**	**50 drivers**
Original Data Set	4898, 16.3 avg	338, 1.12 avg	35, 0.12 avg	1, 0.00 avg
New Data Set	6125, 20.4 avg	347, 1.15 avg	45, 0.15 avg	0, 0.00 avg

3 Approach 2: Pre-emptive shortest wait scheduling with variable number of drivers

3.1 Algorithm Overview

After experimenting with my first approach, I realized that there were two possible optimizations to make. First, knowing that the *maxCommittedTrips* parameter makes little difference in the overall performance we do not need to run the algorithm over a simulated time, but over the individual requests. Second, being greedy by the shortest distance does not always allow for the best results. A better approach would be to assign drivers to passengers by their shortest waiting time. This allows us to put a driver to where a passenger might be before they even make their request, resulting in zero wait time.

The precomputed distances using Dijkstra's algorithm was not modified from the first approach. The passenger data structure also stayed the same. The driver structure now has an additional "currentTime" variable, and the trip structure is no longer in use. The list of passengers is in a queue sorted by request time, as before. For each element in the list, the driver who would pick up the passenger with the shortest wait time was found. This was found by taking the driver's simulated time, adding the time it would take get to the passenger from the driver's current location, subtracting it from the request time, then setting it to zero if the difference was negative. A negative means that case where a driver arrives before a passenger made the request. Once the best driver is found, their current location is updated to the passenger's end location. The driver's current time is increased by the time it takes to go to the passenger, the time it takes to drop the passenger off, and any time the driver needed to "stall" if it arrived before the passenger made the request.



```cpp
// Create graph from parsing network.
const unsigned int N = 50;
CityGraph<N> graph;
graph.parseCSV("./data/network.csv");

// Parse the passengers.
std::queue<Passenger> passengers;
parseRequests("./data/supplementpickups.csv", passengers);

vector<Driver> drivers;
for (int i = 0; i < settings.numDrivers; i++)
{
    drivers.push_back(Driver(i % N));
}

int totalWaitingTime = 0;
while (!passengers.empty())
{
    Passenger p = passengers.front();
    passengers.pop();

    int minWaitingTime = 1e9;
    int bestTimeToPickup = 1e9;
    Driver* bestDriver = nullptr;
    for (Driver& d : drivers)
    {
        int timeToPickup = graph.shortestDistancesFrom(d.currentNode)[p.startLocation];
        int waitingTime = std::max((d.currentTime + timeToPickup) - p.requestTime, 0);
        if (waitingTime < minWaitingTime)
        {
            bestDriver = &d;
            bestTimeToPickup = timeToPickup;
            minWaitingTime = waitingTime;
        }
    }

    // The distance to the dropoff.
    int timeToDropoff = graph.shortestDistancesFrom(p.startLocation)[p.endLocation];

    // If an uber arrives too early, it needs to "stall" until the passenger is ready.
    int stallTime = std::max(0, p.requestTime - (bestDriver->currentTime + bestTimeToPickup));

    // Update the best driver.
    bestDriver->currentTime += stallTime + bestTimeToPickup + timeToDropoff;
    bestDriver->currentNode = p.endLocation;

    // Keep track of how long the passenger was waiting for.
    totalWaitingTime += minWaitingTime;
}

cout << "Total waiting time: " << totalWaitingTime << endl;
```

Figure 9 - Implementation of shortest wait time approach

3.1.1 Complexity

Pre-computing distances has a constant $O(|E||V| + |V|^2 \log(|V|))$ complexity that does not vary with the number of requests. Considering that this only needs to be computed once, the overall time complexity is $O(kn)$, where k is the number of requests and n is the number of drivers.

3.2 Results

The number of drivers was plotted against the total wait time. The exponential behaviour was still observed, but the decay was significantly faster. Using this attempt, only five drivers were needed to reduce the total wait time to almost zero.

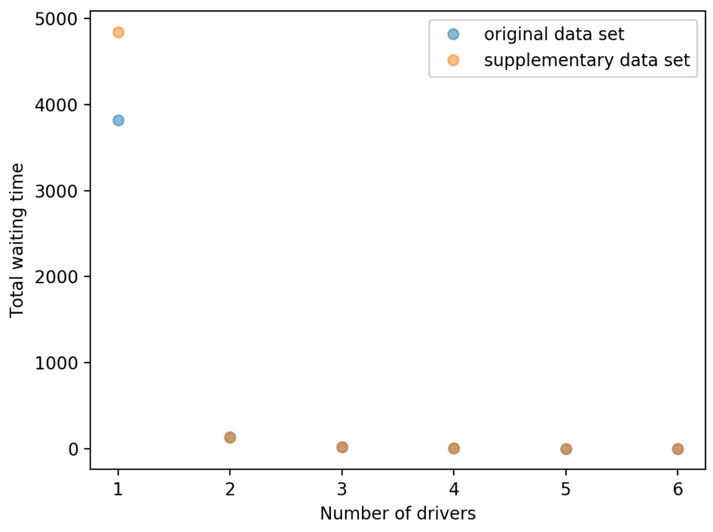


Figure 10 - Plot of wait time from shortest wait time approach

This approach reduced the total wait time with two drivers by a factor of 2.5. To reduce the wait time to almost zero, the number of drivers was reduced by a factor of 10. An interesting point is that if the shortest wait algorithm does not allow pre-emptive scheduling where drivers can only move after request is made, it performs almost identically to shortest distance in Table 1.

Table 2 - Summary of results from shortest wait time approach

	1 driver	**2 drivers**	**3 drivers**	**5 drivers**
Original Data Set	3818, 12.7 avg	133, 0.44 avg	23, 0.08 avg	1, 0.00 avg
New Data Set	4845, 16.1 avg	131, 0.43 avg	20, 0.07 avg	2, 0.01 avg

4 Conclusion

Finding the optimal set of combinations for uber dataset in an NP-hard problem. However, greedy approaches using shortest distance and shortest wait heuristics and Dijkstra's algorithm provided promising results with both data sets. As the number of drivers increased, the total waiting time reduced exponentially. Using two drivers with shortest distance resulted in the average waiting time being 1.12 and 1.15 units of time for the first and second data set respectively. With shortest wait, the average waiting times were reduced to 0.44 and 0.43 units of time. The total wait time was almost eliminated with 50 drivers with shortest distance, and 5 drivers using shortest wait. Shortest distance has a complexity of $O(tkn)$, while shortest wait has a complexity of $O(kn)$. Both require an initial $O(|E||V| + |V|^2 \log(|V|))$ complexity to find the shortest distances from each starting node. The only disadvantage of shortest wait is that it optimally requires pre-emptive scheduling, moving a driver to a passenger's location before they make a request. Pre-emption is disallowed, then shortest wait performs almost identically to shortest distance. Considering that both approaches can be implemented in polynomial time, they provide good solutions that would otherwise require factorial time complexity with brute force.