Uber Assignment

20023798

Eric Leask

2018/11/21

CMPE 365

Contents

[Introduction 1](#_Toc530605905)

[Assumptions 1](#_Toc530605906)

[Description Assumptions 1](#_Toc530605907)

[Algorithm Assumptions 2](#_Toc530605908)

[Key Solution Components & Complexity 2](#_Toc530605909)

[Uber Class 2](#_Toc530605910)

[Pathfinding (Function: dijkstras) 2](#_Toc530605911)

[Scheduling (Function: runCalc) 2](#_Toc530605912)

[Dynamic Memory (Function: dynCheck) 3](#_Toc530605913)

[Experimental: Optimal Nodes 3](#_Toc530605914)

[Results 4](#_Toc530605915)

[Complexity 4](#_Toc530605916)

[Dynamic Memory 4](#_Toc530605917)

[Impact of Increased Number of Drivers 4](#_Toc530605918)

[Going Forward 5](#_Toc530605919)

[Figure 1-Mapped network 1](#_Toc530605926)

[Figure 2-Mapped network with two identified groupings of nodes. Left most is defined as GroupA and right most defined as Group B 3](file:///C:\Users\Eric%20Leask\Documents\Eng%20Year%203\365-Algorithms\FINAL%20ASSIGNMENT\Uber_20023798_EricLeask.docx#_Toc530605927)

[Figure 3-Gains and Losses of utilizing the experimental system versus without 4](#_Toc530605928)

# Introduction

Given the static data of street level transit times, and a sequence of requests for service for an Uber style pickup service. An algorithm is produced to determine the total length of time that passengers spent waiting with the purpose of optimizing this value through scheduling. At its core this is a scheduling algorithm with varying task length (wait time) depending on what resource (Uber car) the task is given too. The algorithm has two levels. The main algorithm is a FIFO (First in First out) scheduling algorithm. When deciding which Uber car will be chosen to handle the request, Dijkstra’s is implemented to calculate the distance between the car and the passenger pickup point.

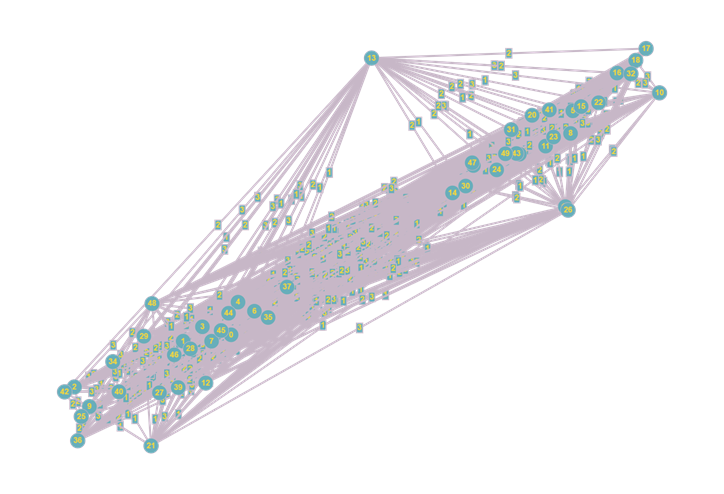


Figure -Mapped network

# Assumptions

Outlined here are the assumptions from the description of the problem and assumptions made during the production of the algorithm.

## Description Assumptions

* *“Uber drivers fulfill requests in the order of their start time stamp.”* A driver may only leave to resolve a request if all previous requests are completed or in the process of being completed.
* *“It then takes the passenger to the finish location they requested; and then fulfills the next outstanding request.”* The driver may only deal with a single request at once. It can not deal with a second request in the process of dealing with the current request

## Algorithm Assumptions

* It is expected that the list of requests given is complete A.K.A data entered does not include blank data or non-numeric values. The data is also expected in the same format as the example formatting
* Drivers cannot travel to the location of a future pickup based on the knowledge of that future request. This is under the assumption that this is a real-life situation where you would not know when and where future requests were happening.
* All connections between nodes are assumed to be bidirectional. No statements were made otherwise to counter this assumption.

# Key Solution Components & Complexity

## Uber Class

The uber car class was defined for the purpose of bundling together the key values depended to each uber car in a neat format. Within this class the two variables, start and end, describe the starting and ending nodes of the most recent request the vehicle dealt with. The variable dropTime contains the time at which the most recent request will or has completed.

## Pathfinding (Function: dijkstras)

A\* and Dijkstra were considered when choosing a path finding algorithm. A\* was originally considered to be implemented due to its improved computation performance over. Note that A\*’s benefits over Dijkstra’s are strictly performance wise and does not find a more optimized path, merely the same route faster.

A\* fundamentally is an extension of Dijkstra’s. A\* utilises a heuristic algorithm to guide the Dijkstra. To “guide” the Dijkstra’s algorithm, the heuristic algorithm affects the order of the heap which Dijkstra reads nodes from to expand the path of its node search. In Dijkstra’s we search based on current minimum weight to be reached node until we arrive at the location. With A\*, it chooses the next node based on the combination of the minimum weight (like Dijkstra) and the value of the heuristic algorithm which represents the distance to the end node. The estimate of the total path keeps the algorithm away from solving for nodes which would be moving away from the goal and thus removing inefficiencies.

The choice of Dijkstra’s over A\* was made due to the complexity of producing a heuristic algorithm for the network. Commonly, the heuristic algorithm involves using the direct distance between the current and final node on a Euclidean(x,y,z) style map. The network of nodes could not be formatted appropriately to this form. This leads to the conclusion of using Dijkstra’s for path finding.

The complexity of this Dijkstra’s implementation is O(2v2) which reduced to O(v2). This is due to the usage of an adjacency matrix and an unsorted array.

## Scheduling (Function: runCalc)

The logic of the scheduling is quite simple. The scheduling by the description of the problem, specifically requests must be handled in the order they occur. This makes this a FIFO implementation. We call a time function for each uber vehicle. The functions purpose is to determine the wait time to arrive at a request’s location. The function uses Dijkstra to get the time to go from the end location of the vehicles last trip to the request’s location. Then it checks if the vehicle his still busy with a request. If it is, it takes the difference between when the cars current request will finish and the time of the new request. This is added to the time function. A loop iterates through all cars and stores which car has the minimum weight to reach the passenger. This vehicle’s start, end, and dropTime values are updated to match the requests values. Then scheduler then moves on to the next request.

The function runCalc’s greedy FIFO scheduling algorithm component is defined as O(n) where n is the number of requests. This is ignoring the call of Dijktras inside it.

## Dynamic Memory (Function: dynCheck)

To improve runtime, routes chosen to be traveled are stored in a list. When Dijkstra is executed it will check the list for previous found paths bidirectionally. In a situation where the dynamic memory can be utilised, it will improve the pathing finding component’s run time to O(n).

dynCheck iterates through all values in the list dynMemory to determine whether the route has been taken before ignoring direction. This search algorithm takes O(n) to compute where n is the number of requests. In this scenario runCalc’s complexity would be O(2n) which simplifies to O(n)

## Experimental: Optimal Nodes

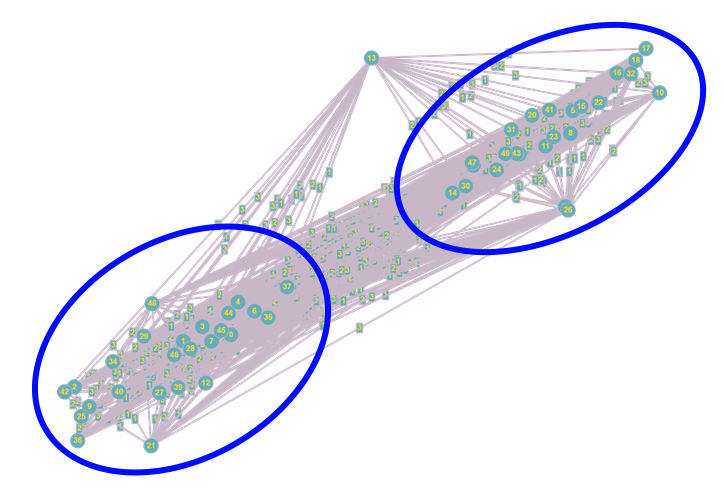
Note that this is not considered to be the official solution due to the volatility of the outcomes. Mapping the network shows that excluding a single outlying node, there are two large bunching’s of nodes grouped by interconnectivity. A situation could occur where both cars are located in one of the two bunches and receives a request from the other bunch. The reduced connectivity between the two groups could result in an extended wait time. To combat this possibility the resulting logic was implemented. When Vehicle 1 begins a request, if Vehicle 2 is not busy it will move to a chosen node in the group Vehicle 2 is not traveling too. This hypothetically can benefit the run time since the there will be a vehicle locate at a node with a low weight distance to a high number of nodes.

Figure 2-Mapped network with two identified groupings of nodes. Left most is defined as GroupA and right most defined as Group B

The node chosen for each group is based on minimizing two factors, the number of nodes it does not connect to, and the number of routes weighted 3 units. Node 12 in the left group was chosen due to 60% connectivity to all other nodes and of those nodes only 16.7% of routes to other nodes had a weighted cost of 3. The chosen node for the right group was Node 26 with a 50% connectivity and 16.7% path weight of 3. It’s noted that this experimental system is only designed for two vehicles and the given network.

Figure -Gains and Losses of utilizing the experimental system versus without

|  |  |  |  |
| --- | --- | --- | --- |
| File | Total Wait Time Norm Sys. | Total Wait Time Experimental Sys. | Gains |
| supplementpickups.csv (supplied) | 617 | 615 | +0.3% |
| requests.csv (supplied) | 618 | 589 | +4.9% |
| requests2.csv (custom) | 551 | 571 | -3.5% |

In practice, the performance is dependent on the given data set. Sets where requests are dispersed farther apart will benefit the performance and vice versa. If nodes which are part of the 16.7% of nodes with a weight a 3 from the chosen nodes, then the performance suffers and vice versa if not within the 16.7%.

# Results

## Complexity

The final complexity of the algorithm is O(nv2) and Ω(nv). The schedule algorithm runs n times, once for each request. For each request, Dijkstra’s is called twice (this 2 is a constant which is removed from the final complexity) to calculate the time to reach the requested location. This takes O(v2) in the worst-case scenario. In the best-case scenario instead of using Dijkstra’s, the algorithm finds the path in the dynamic memory built as the code runs which is O(v).

## Dynamic Memory

Dynamic memory reduces run times by a factor of approximately 10. When Dynamic Memory is either enabled or disabled the run time increases linearly as the number of vehicles increases. For reference, with 2 cars and dynamic memory enable the run time is 0.141s and 1.176s without dynamic memory.

## Impact of Increased Number of Drivers

The implementation allows for the number of drivers to be varied. As the number of Uber cars increase the wait time reduces. Increasing car numbers reduces the probability of a higher travel time to pick up a passenger due to the vehicles being spread farther apart on the network. The total wait time’s minimum is 306 at 10 vehicles. When measure performance, as the number of vehicles increased, the run time increased as well.

This minimum is due to all cars starting at Node 0. Once a car is called for a request it remains at the drop off location. Once there are 10 cars disperse in the network, the distance the cars are from any request is closer than all the extra cars waiting at Node 0. If the cars began dispersed in the network, the total wait time should hypothetically minimize to 0 around 50 cars in a best-case scenario since there is now one vehicle for every node as long as vehicles shuffle to keep all nodes covered.

# Going Forward

With more time the complexity of the algorithm and wait time could be reduced. Reducing the complexity could be done multiple ways. Sorting the dynamic memory would reduce the search complexity from O(n) to O(logn). Sorting individual values into dynamic memory would take O(logn) each time. Dijkstra’s may be improved by converting the adjacency matrix to an adjacency list. This would reduce this component to O(vlogv). The unsorted array could be substituted for a Fibonacci heap which has a complexity of O(1) for minimum searches and insertion of values.

To improve the total wait time the concept of a heatmap could be implemented. The concept involves using previous requests to predict future requests. The previous requests could be recorded and nodes or areas of nodes with higher request rates could be assumed to have requests occur there in the future. An available car could travel to these locations and wait when not busy.