# Aadiyat Ahmad CMPE 365 Project – Uber

## Problem Definition and Assumptions

The street network of a city is represented by a graph, where the nodes of the graph are locations in the city and the edges are the time it takes to travel between locations.

Two Uber drivers are available to fulfil requests. A sequence of requests for service are given. These include a start time, a starting location and the end time.

The goal is to assign drivers to requests to minimise the total waiting time where the waiting time is the difference between the time at which a driver arrives at the starting location of a request and the start time of that request.

For realism, it assumed that not all requests are known to the Uber drivers beforehand. That is if the current time is *t* the Uber drivers only know about requests with a start time at or before *t*. In this scenario the start time of a request is the time at which a passenger sends the request to Uber, and all passengers want to be picked up immediately.

It is also assumed that there is no way in which to reliably predict when and where future requests will take place. Therefore, at any given time *t* Uber can only optimise assignments of drivers to requests for the requests that are known at *t*.

## Overall Program

### City Street Network

The files containing the static data representing the city street network. Dijkstra’s algorithm is used to find the minimum time it takes to travel between any two nodes and the results are stored in a look up table. This way there is no need to recalculate the minimum travel time between two nodes when calculating the time at which a driver will arrive at the starting location of a request. The minimum time required to travel to the start location of the request can be accessed from the look up table in constant time.

Calculating the minimum travel times between all nodes in this manner may incur some additional costs, if there are pairs of nodes between which no one ever travels. But it saves more computation cost by avoiding recalculating the minimum travel times between nodes that are visited often.

An alternate approach would have been to start with an empty look up table, calculate minimum travel times between nodes for a request and enter the data in the look up if the data was not already present, or access the data from the look up table if it was present. However, pre-calculating all pairwise minimum travel times is simpler. It is also very unlikely that in a real city there are many nodes that never see any travel between them.

Storing a minimum travel time look up table for 50 nodes, where the entries are integers only requires 1000 bytes of memory. And it is not unreasonable to assume that a ridesharing service might store the precalculated data.

The nodes in the city go from 1 to 50, but the indices of the lookup table go from 0 to 49. Therefore, whenever the lookup table is indexed for nodes *u* and *v* it is indexed with the indices *u-1* and *v-1*.

### Requests

The file containing the requests data are read into a queue called futureRequests in the order of the start times. This queue represents the requests that have not yet been “received” by Uber.

Each request is represented by a struct. Each instance of a request has the start time, the starting node, the ending node and the time spent waiting. The values of start time, start node and end node are initialised with the values read from the requests files. The wait time is initialised to -1.

### Drivers

Each driver is represented by a struct. Each instance of a driver has its destination and the timeAtResponse, which is the time at which the driver can respond to a new request (i.e. the time at which the driver can begin travelling towards the request start location). If a driver is currently serving a request then the destination is the end location of the request, and the timeAtReponse is the time at which it will finish the current request. If the timeAtReponse is less than or equal to the current time, it means that the driver has completed the last assigned request and is currently idle. If a driver is idle, then the destination is the driver’s current location and the timeAtReponse is the current time.

### Main Loop

A while loop is used to simulate the passage of time. Each iteration of the loop representing the passing of one time unit. The current time starts at 0 and is incremented by one at the end of each iteration of the loop.

To simulate requests being “received” the program keeps track of the time at which the next response will be sent. This is the start time of the request at the front of the futureRequests queue.

Within each iteration of the loop the timeAtReponse for each idle driver is updated to the current time.

All requests that have start time at the current time are popped from the futureRequests queue and added to a vector, pendingRequests. If new requests were added to pendingRequests then a flag is set to true.

Assignments of pendingRequests to drivers is done using a Branch and Bound algorithm. The branch and bound algorithm accepts as parameters the vector of pending requests and the state of both drivers and returns a vector, optimalDecisions.   
optimalDecisions is the same length as pendingRequests each element of optimalDecisions corresponds to an element of pendingRequests. The entries optimalDecisions indicate which driver is assigned to the corresponding request.

If a driver is idle, it serves the first request in pendingRequests that is assigned to it, if there are any. Before the driver serves the request, if there were new requests added to pending requests since the last time the optimal decisions were calculated, the optimal decisions are calculated again to account for the new pending requests. This way the assignment of drivers will always minimise wait time for the requests that are known.

When a driver is assigned to a request, the wait time for the request is calculated. The request is removed from pendingRequests and added to a queue of completed requests.

At the end of the iteration the current time is incremented

### Total Wait Time

The total wait time is calculated by summing the wait times of all requests in the completed requests queue.

## Branch and Bound

A branch and bound algorithm is used to assign drivers to requests to minimise the waiting time. The objective measure of cost being minimised by the branch and bound is obviously the waiting time.

When the assignment of requests to drivers needs to be recalculated there are *n* pending requests and 2 drivers. Therefore, there are 2n ways of assigning requests to drivers. The decisions to assign requests to drivers can be represented as 2-ary tree where successive levels of the decision tree represent successive requests in order of their start time. The branch and bound method explores this decision tree, ignoring subtrees where the paths through the subtree will not lead to a feasible solution.

An alternate way of assigning requests to drivers is to use the greedy approach. Since the requests must be fulfilled in the order of the start time, the greedy method would involve assigning to each request the driver that can arrive at the request start location first. But the greedy approach is not guaranteed to minimise the waiting time, since the decision made for each request does not consider the subsequent requests. See the Greedy Algorithm Counterexample section for details.

The branch and bound method does consider all the pending requests and produces the set of decisions that minimises the total waiting time.

The implementation of the branch and bound algorithm performs a breadth-first traversal of the decision tree. Each instance of a decision node tracks its level in the decision tree (i.e. which request it is a decision for), the waiting time upper bound of all decision paths going through that node, the waiting time lower bound of all decision paths through that node, a vector of the sequence of decisions leading up to the node and the state of each driver due to that sequence of decisions.

In the initialisation stage of the branch and bound, a dummy node is pushed to a queue of decision nodes. The dummy node does not represent a decision, but simply is simply the root of the entire decision tree. It initialises the decision tree with the state of each driver at the time the branch and bound function is called.

A vector called optimalDecisions stores the assignment of drivers to request in the optimal decision path. optimalDecisions is the same length as pendingRequests each element of optimalDecisions corresponds to an element of pendingRequests. The entries optimalDecisions indicate which driver is assigned to the corresponding request.

An overall upper bound is calculated using the greedy algorithm as described above.

While the queue of decision nodes is not empty, the first node is popped. If *u* is the last node of a full solution, then nothing is done. Otherwise, from each node *u* the next request can be assigned to one of two drivers. These decisions are represented by nodes *v* and *w*. For each of *v* and *w* the decisions made so far is updated to include the decision made at that node. The state of the drivers resulting from the decision is updated, and the lower bound and upper bound are calculated.

The lower bound is simply the sum of the wait times of the requests in the path leading up to the node (this assumes the best case, which is that the wait time for all subsequent requests is 0). The upper bound is found using the greedy algorithm described earlier, starting from that node.

If the lower bound at a decision node is higher than the overall upper bound, then there is no feasible solution through this node and it is ignored. Else the node is pushed into the queue of decision nodes to be considered again.

If the node is the last node of a full solution and the upper bound at the node is lower than or equal to the overall upper bound, the overall upper bound is made equal to the upper bound at that node and optimalDecisions is made equal to the series of decisions leading up the node.

Then the loop repeats until the queue of decision nodes is empty. If the queue is empty, the optimal path through the decision tree has been traversed. Therefore, at the end of the branch and bound, optimalDecisions will contain the optimal assignment of drivers to requests.

### Some Observations

In branch and bound it is possible that if driver 0 is assigned request *a* and driver 1 is assigned request *b,* and *a* has a later start time than *b*, driver 0 will serve request *a* before driver 1 serves *b*. But each driver will serve the requests assigned to that driver in the order of the start times. So the design requirements are still satisfied.

If the program did not simulate the passage of time and took a less realistic approach by assuming that all the requests are known at the start of the program, then the program would be simpler. Instead of a while loop simulating time and optimising for pending requests if there are any new pending requests, there would only be one call to branch and bound that takes all requests into consideration and returns the optimal assignments.

## Complexity Analysis

### Dijkstra’s Algorithm

Let |V| be the number of nodes, and |E| the number of edges.   
The candidate and reached sets of Dijkstra’s algorithm are both implemented as C++ sets. Insert and Erase in C++ sets are both logarithmic time operations, O(log |V|).  
The edges are represented by an adjacency matrix, so edge weights between any two nodes can be accessed in constant time, O(1). Checking if two nodes are neighbours (i.e., checking a node exists) is a linear time operation, O(|V|).  
The minimum distance to each node is stored in a vector. Searching the minimum distance vector is linear, O(|V|).

Finding the node with the minimum distance involves finding the minimum distance in the distance vector and then removing a node from the candidate set and adding it to the reached set. So the total complexity is O(|V| + log|V|).  
Finding and updating neighbours is O(|V|).  
This is repeated for every node in the graph. So the total complexity is O(|V||V|+ |V||V| + |V|log|V|) = O(|V|2)

The complexity is dominated by the fact that it takes O(|V|) time to check if two nodes are neighbours, and this is done for |V| nodes.

To calculate the minimum travel time lookup table, Dijkstra’s algorithm is used |V| times. Bringing the total complexity to O(|V|3)

The complexity of Dijkstra’s algorithm, however, has little influence on the program as it is used simply to calculate a look up table, and does not affect the time need for Uber to examine the pending requests and calculate an optimal assignment of drivers to requests.

### Branch and Bound

In the worst case of branch and bound all the possible paths of the decision tree must be explored. If there are *n* pending requests there are 2n possible paths. However, it is expected that the complexity will be better than that as unfeasible paths will be pruned before they are explored.

The number of times Branch and Bound is called within the main loop of the program is not related to the number of requests, but rather how the requests are spaced out in time. For example, if all the requests have the same start time, then there is only one call to branch and bound. If the requests are spaced out, then there are more calls to branch and bound as new pending requests arrive, but in that case, at any given time, there will be fewer pending requests.

## If Number of Drivers Was an Input

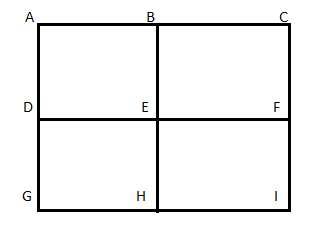
If the number of drivers was an input, the parts of the code where identical steps must be taken could be implemented in a for loop.

The total wait time is expected to be inversely proportional to the number of drivers.

## Greedy Algorithm Counter Example

Here I provide an example where the greedy algorithm does not find the optimal solution.

Consider the following graph where all the edges have weight 1.



The adjacency matrix for the graph is follows



Using Dijkstra’s algorithm to find the minimum distance between every pair of nodes produces the following look up table of minimum distances



Suppose the following requests are pending



Driver 0 and Driver 1 are both idle. Driver 0 is at node F and Driver 1 is at node H. Using the greedy algorithm, Driver 0 can respond to the first request quickest and so the first request is assigned to Driver 0. Driver 0 will arrive at the start location at time 1 and complete the request at time 5. The wait time is for the first request is 1.  
Driver1 can respond to the second request quickest and so the second request is assigned to Driver 1. Driver 1 will arrive at the start location at time 4 and complete the request at time 5. The wait time for the second request is 3.  
Driver 0 can respond to the third request quickest and so the third request is assigned to Driver 0. Driver will arrive at the start location at time 6 and complete the request at time 7. The wait time for the third request is 4. The total wait time is 8.

However, if the Driver 0 was assigned to both the first and second request, and Driver 1 was assigned to the third request, which is the result of the branch and bound algorithm, the total wait time is shorter.  
Under this assignment, Driver 0 will arrive at the start location of the first request at time 1 and complete the request at time 5. The wait time is for the first request is 1.   
Driver 0 will arrive at the start location of the second request at time 7 and complete the request at time 8. The wait time for the second request is 5.  
Driver 1 is already at start location of the third request and so the wait time for the third request is 0. The total wait time is 6.

Therefore, the greedy approach does not guarantee the optimal solution.