Computer Science Large Practical 2015 - Minibus Simulator
The University of Edinburgh

A study on the effect of fleet size and allowable delays on a minibus transport system using a stochastic simulator

21/12/2015
Gavin Waite
S1208506

1. Overview of how the system operates

The minibus simulator is a stochastic simulator which models an on-demand style public transport system. Requests enter the system at time intervals drawn from a random exponential distribution, at which point the simulator must either report a scheduled pickup time for that request or announce that the request cannot be accommodated. Each request is made up of a random origin and destination stop and a desired pickup time which is drawn from a second exponential random distribution. The main task of the simulator is to decide which, if any, of the fleet of minibuses can satisfy the incoming requests the fastest and most efficiently.

To facilitate in the planning and routing of the buses, the time and fastest route between all pairs of stops are calculated before the simulation begins and stored in a two dimensional lookup matrix. This can then be checked when needed by the scheduling function.

The buses, when they are not idle, each have a queue of stops that they are planning to visit next. Stops in this queue that are, or lie before, pickup stops (a stop where a passenger is scheduled to be picked up) are permanent as the bus is obliged to follow through with its promised pick up time. Stops after all these pickup stops however, are free to be re-organised as new requests come in if it allows for a more efficient routing.

When a request comes in and is processed, a function is called to determine the time to reach both the origin and destination stops for that request for all buses in the system. The time to origin value is then used to schedule the bus if it is chosen and the bus with the lowest time to destination (assuming the time to origin is within the requirements) is the one which is picked.

The buses can be in one of four states (1,3,5,6) when the function is called which determines how the time values can be calculated:

1. The bus is idle at a stop
2. The bus is non-idle
    3. The origin stop of the new request lies after the last pickup stop in the bus's queue of stops to visit
    4. The origin stop of the new request is not present or lies before the last pickup stop in the bus's queue of stops to visit.

5. There is at least one pickup stop in the bus's queue
6. There are no pickup stops in the bus's queue

Therefore, the first task of the simulator is to simply determine which one of these four states applies to the bus. It can then apply the appropriate method to calculate the required travel times.

If the bus has no stops in its queue and is therefore idle (State 1) then the time to the origin and destination stop can be simply calculated from the shortest path lookup matrix, remembering to allow for the embarking time of the passenger.

If the bus is not idle and will pass through the origin stop after all scheduled pickup stops then it is in State 3. In this case the system must check if the new destination stop is also present (after the origin) in the queue. If so then all that is needed is to consult the shortest path matrix to get the time to the two stops. If not then the time to the destination can be found as the time to get to the final stop in the queue, plus the time from there to the destination.

If the bus is in State 5 above, then the new origin could be added after the current final pickup stop. The time is therefore calculated as the time to the last pickup stop plus the time from there to the new origin. The bus should then return to complete the drop offs that it had been planning to do before heading to the new destination stop.

If there are no pickup stops then the bus is in the final State 6 above. In this case the bus has passengers on board that are due to be dropped off but no more scheduled to pick up. The decision was made for the bus to be allowed to complete its drop offs before adding another passenger. This was done in an effort to avoid extremely long trip times where a passenger's disembarking would continually be delayed by detours to new pickup stops. Therefore the time to origin and destination are calculated as the time to the final stop plus the time to them.

Once the fastest bus is known, if the time to origin is less than the desired time plus the maximum admissible delay, it is then scheduled and it's queue updated to reflect the new route. If the time to origin is less than the desired time then a delay is also queued for once the bus reaches the stop before it seeks to board the passenger.

During normal operation, the simulator simply increments the time in a loop (until the specified maxTime) until the next event occurs. An event is defined as any time a bus reaches or leaves a new stop, when a bus's scheduled delay ends, when a passenger is boarded or disembarked or when a new request is processed. All but the delay event are official events which are required to be output to the console, along with the changes to occupancy that come with boarding or disembarking.

Once a bus reaches a new stop, various details are checked. Firstly, any disembarking passengers are let off at that stop. Then the system checks if there is a scheduled delay for that bus at that stop and begins waiting if necessary. After the delay completes, the system tries to board a passenger (if appropriate). If the current time is not exactly equal to the

scheduled time then an error is thrown. This keep the simulation valid. After all boarding has completed, the system checks if there are any other stops to go to. If so, then the bus is started moving again, otherwise it becomes idle.

Throughout the simulation, individual passengers, buses, requests and trips are tracked in order to generate some useful statistics. The statistics are reported at the end of the simulation, as specified by the requirements. These can be analysed while varying one of the input parameters to investigate its effects (see section 4).

2. Design choices

Throughout the design and implementation of the simulator, several important decisions had to be made. The most critical are detailed in this section.

My simulation holds the future routes of all minibuses so that the scheduling decision for each new incoming request can immediately be evaluated. Another possible approach route would have been to allow the requests to be evaluated at any time before the desired departure time, hypothetically queued at each origin stop, but my interpretation of the requirements was that the decision was triggered as soon as the new request enters the system.

My simulation increments in time intervals of one second, instead of computing the lowest delay and then incrementing all delays by that amount. By keeping my main simulation loop to a minimum, simply decrementing all countdowns and then checking if they are zero, all non-event cycles operate very quickly. Thus the performance decrease in this simplification is minimal. However, it would likely still be a worthwhile improvement, if the necessary changes were made to support it.

Requests and trips that run over the maximum boundary *stopTime* had to be carefully considered when tracking the statistics. If a request enters the system with a desired departure time or scheduled departure time after the simulation will have ended it is still scheduled as normal and counted for the *percentageOfMissedRequests* statistic. I felt that this was the fairest option as the requests were still technically generated within the allotted time and the simulator would have been able to satisfy them. For the statistics that require individual passenger trips to be tracked, I solved the end time issue by only incrementing the statistics at the instant when that passenger disembarks. Thus, any ongoing trips will not impair the statistics.

If the bus capacity ever becomes maximum on the last pickup, then there is a planning decision to be made. When attempting to satisfy a new request, if the system is trying to add the route to the new origin after the last pickup but that pickup takes the bus to maximum capacity then there are two options. The first would be to instead try and add the route after the next drop off stop as there would now be available seats on the bus. The second, and the chosen option, was to simply disregard that as an option and check instead the time after the bus has dropped off all passengers. This will take a long time (routing between and dropping off the maximum number of passengers) so will rarely be successfully chosen. However if the first option had been taken then, given a high enough

request rate, a bus could in theory get to maximum capacity and then just alternate between picking up and dropping off passengers, with no guarantee of passengers ever being let off.

3. Model

The core of the simulation model is a globally scoped array *buses[noBuses]* of *Bus* struct objects. The components of this struct are shown below:

```
typedef struct {
    int ID;
    int location;
    bool inTransit;
    bool idle;
    int timeUntilNextStop;
    int capacity;
    Queue nextDestinations;
    Passenger *passengers;
} Bus;
```

Each bus is assigned an ID which matches its position within the *buses* array. This is used by the system when passing a bus around functions so that the physical *buses[bus.ID]* can be changed in memory. The current *location* is stored for all buses while stationary at a stop. While moving between stops *inTransit* is set high and the *location* is set to -1. The *timeUntilNextStop* is the duration of the current transit and is incremented by the main simulation loop. The *capacity* stores the number of passengers currently on board the bus and the *passengers* array holds information about these, and other scheduled future occupants. The *nextDestinations* is my implementation of the queue of future stops that will be visited by the bus. These custom structs are shown below:

```
typedef struct{
    FutureStop *stops;
    int count;
    int first;
    int last;
} Queue;
```

```
typedef struct{
    int stopNumber;
    int arrayIndex;
    int capacityAtStop;
    int passengersToPickUp;
    int passengersToDropOff;
    int delay;
} FutureStop;
```

```
typedef struct {
    bool active;
    bool pickedUp;
    int ID;
    int bus;
    int destStop;
    int schedTime;
    int travelTime;
    int minTime;
} Passenger;
```

Queue is my implementation of the standard data structure of the same name. It has attributes *first, last* which store the array index in the *stops* array of the first and last element of the queue respectively and a *count* of the number of current elements.

The *stops* are each of type *FutureStop*. These are used so that the system can track the parameters of the bus and its actions into the planned future. The *stopNumber* refers to the number of the stop, as expected. The *arrayIndex* refers to the actual position of that stop within the queue from zero to *count*-1. The *capacityAtStop* defines the capacity that the bus will have at that stop if it follows the planned route. The *passengersToPickUp* and *passengersToDropOff* detail any planned passenger pickups and drop offs that are due to be made at that stop. Finally, the *delay* element contains the number of seconds that the bus is scheduled to wait at that stop before attempting to embark a passenger.

The *Passenger* type exists in order to facilitate the tracking of individual passengers through the system. Once a request is made, an *active* passenger is added to the *passengers*

array within the chosen *Bus.* The included details are which *bus* it was assigned to, the *destStop* where it is to be dropped off and the *schedTime* for pickup. When the bus reaches a stop where it is due to pickup a passenger, the system checks for any disembarks first (as normal), and lets those passengers off, then any delays and finally attempts to board the passenger. Once the passenger movement is complete, the system tries to work out which *Passenger* was just boarded. Some verification is granted here as the current *simTime* minus the *boardingTime* is compared with all *Passenger schedTime*s and an error is thrown if the passenger is not found. If it is found, then *pickedUp* is set to true and the *minTime* is set to the shortest time in seconds between the pickup and drop off stops. The *pickedUp* remains true until the passenger disembarks and during this time, the *travelTime* is incremented every second. Upon reaching the disembark stop, if multiple *Passenger*s are due to be getting off, the one with the lowest ID (ie. The one scheduled earlier) is chosen to get off first. At this point *active* is set false and the *passengers* array shifting along to fill the space. The *travelTime* and *minTime* are then added to accumulators for use in the overall statistics.

4. Testing

As mentioned in the READ_ME, I had a serious issue with transferring my simulator to an environment outwith the XCode IDE related to my handling of the global 2D arrays: *nextHopFromTo, shortestLengthFromTo,* and *passengersMoving*. XCOde uses the Apple LLVM 7.0 compiler which successfully compiles my project unlike GCC. Upon realisation of this issue, I had left myself not enough time to find a solution which is regrettable. All following tests are from the working console within XCode itself.

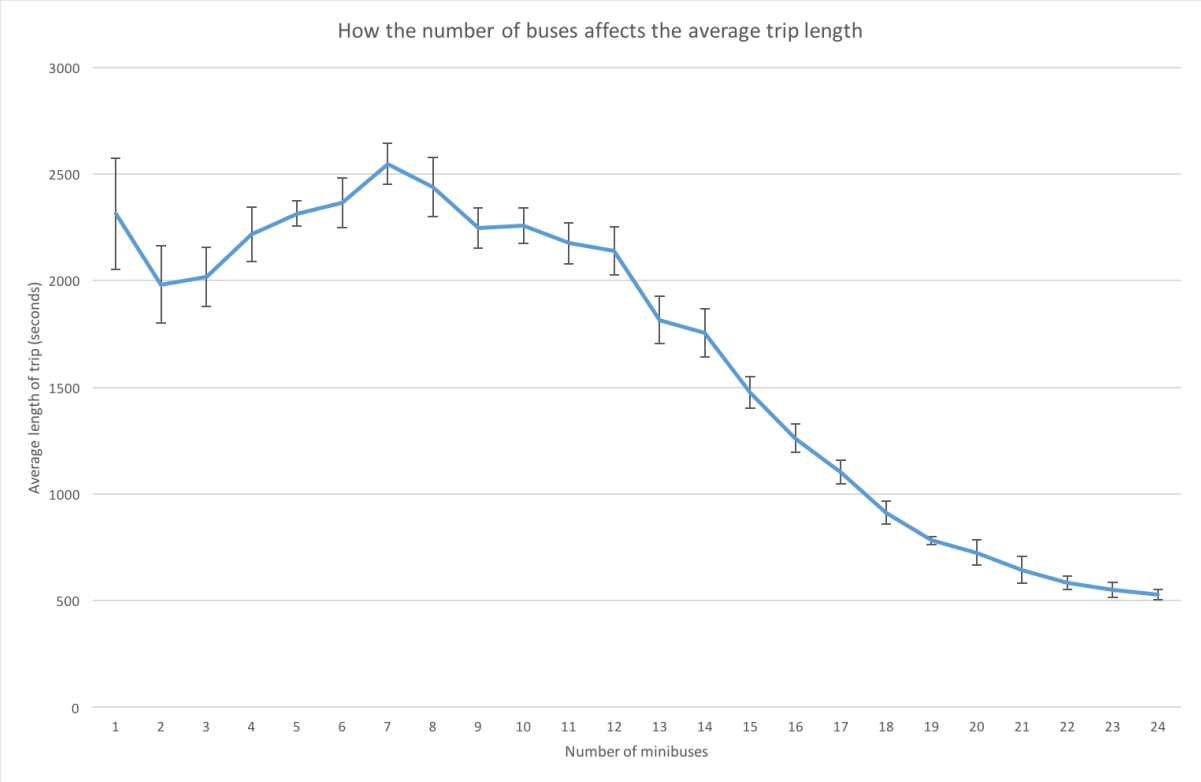
In order to test the system, various input files were used. The input files and the corresponding console output are located in the *TestFiles* folder.

Throughout the simulation, the current state is checked for consistency with itself if ERROR is set to true and the number of errors is output at the end of the simulation

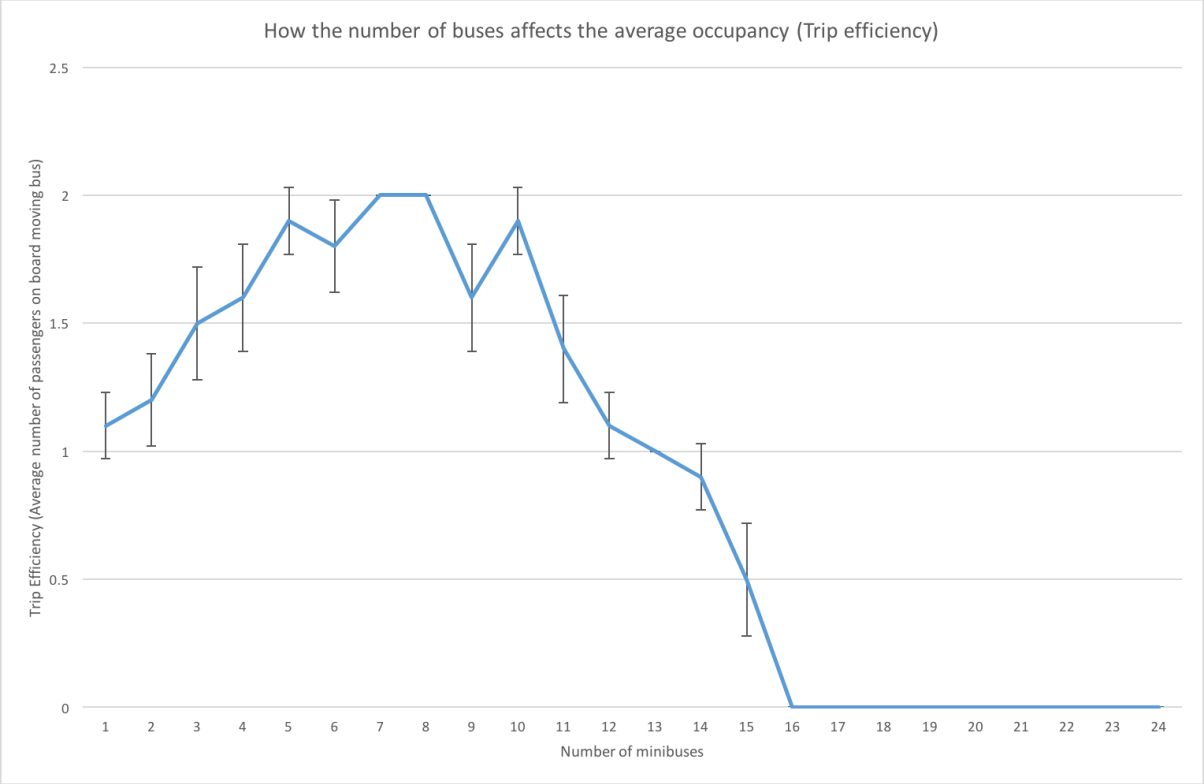
5. Experiments

I used the originally provided input script to explore the effect of **experiment**ing with the number of buses in the fleet and then with the maxDelay parameter. All data produced was produced within the executable program. The ANALYSIS flag is used to repeat the input simulation 10 times and calculate the mean, standard deviation and confidence interval for every statistic for every experiment. The raw data is included alongside the graphs in the *Graphs* folder. This was then plotted using Excel.

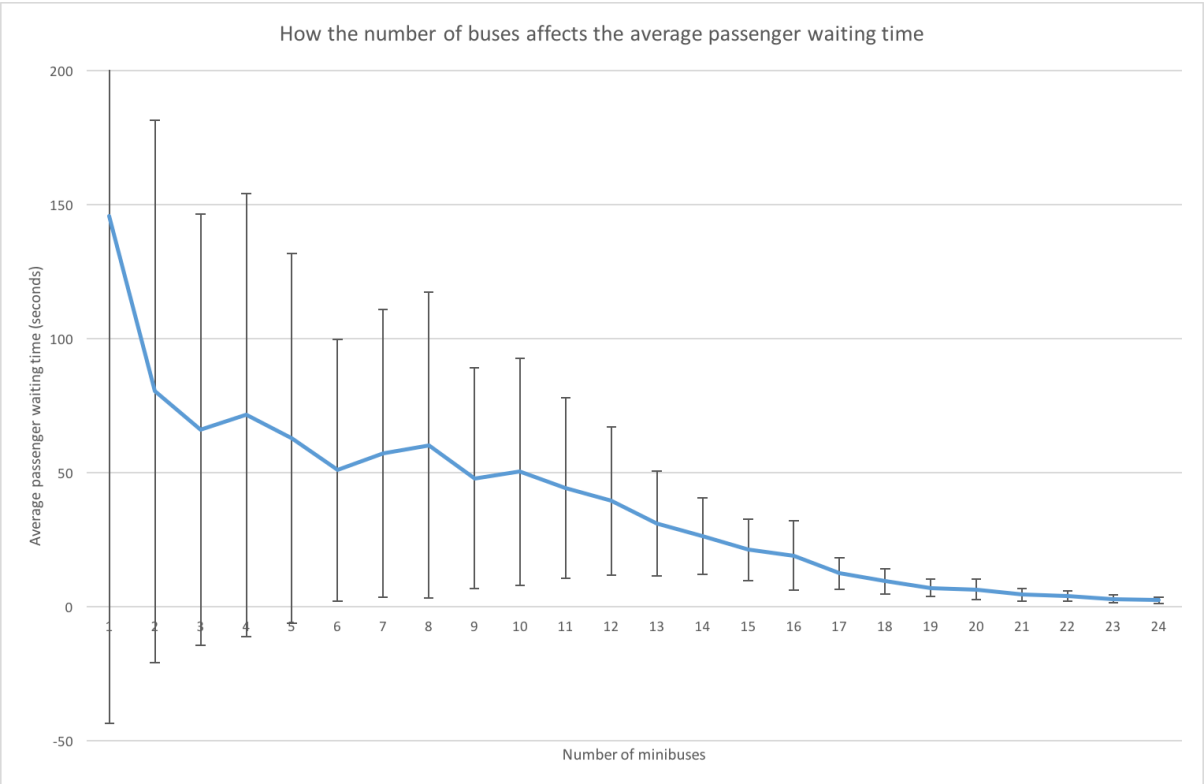
a. Testing the effect of the number of buses



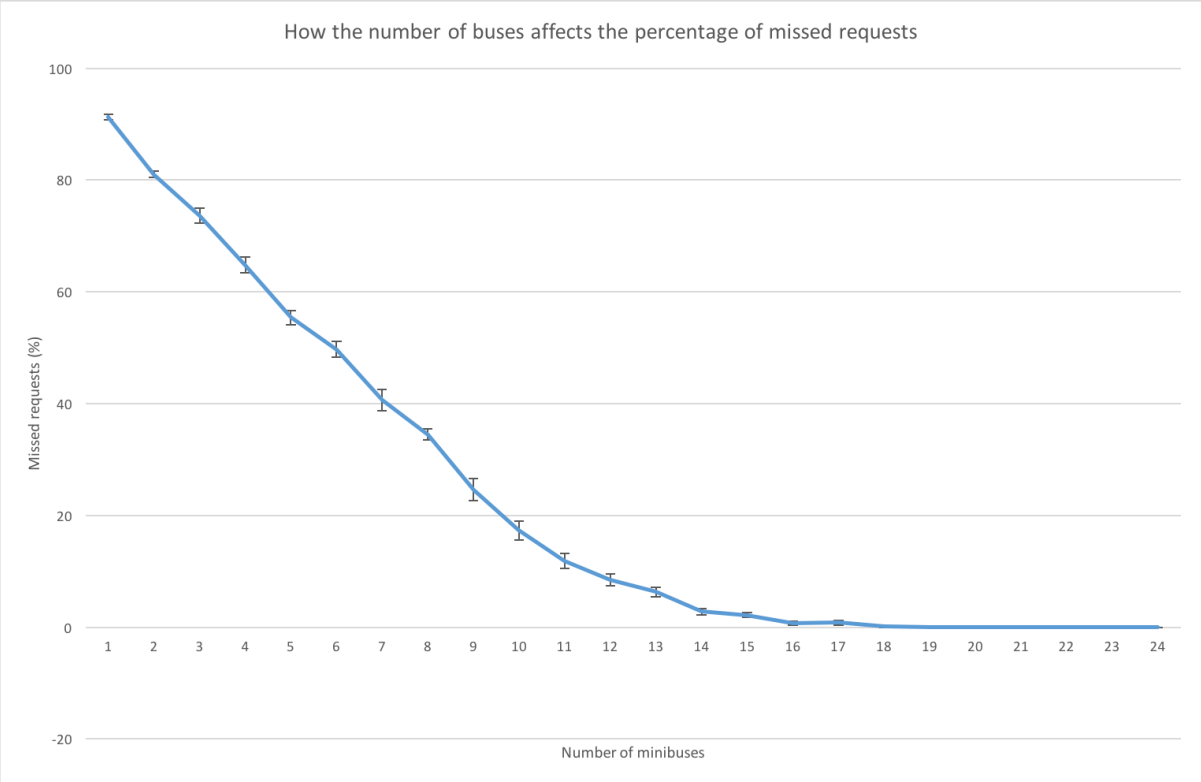
From this we can see that the average trip length in the simulation decreases as minibuses are added. This makes logical sense, more minibuses mean that there will be more idle buses which will typically be the fastest to satisfy a request. Thus there will be fewer instances of buses picking up another passenger while on route. It also suggests that there is little benefit in terms of average trip length until the number of buses is over 12. This might indicate that beyond that amount, the routes are saturated by buses and they will be used less efficiently.



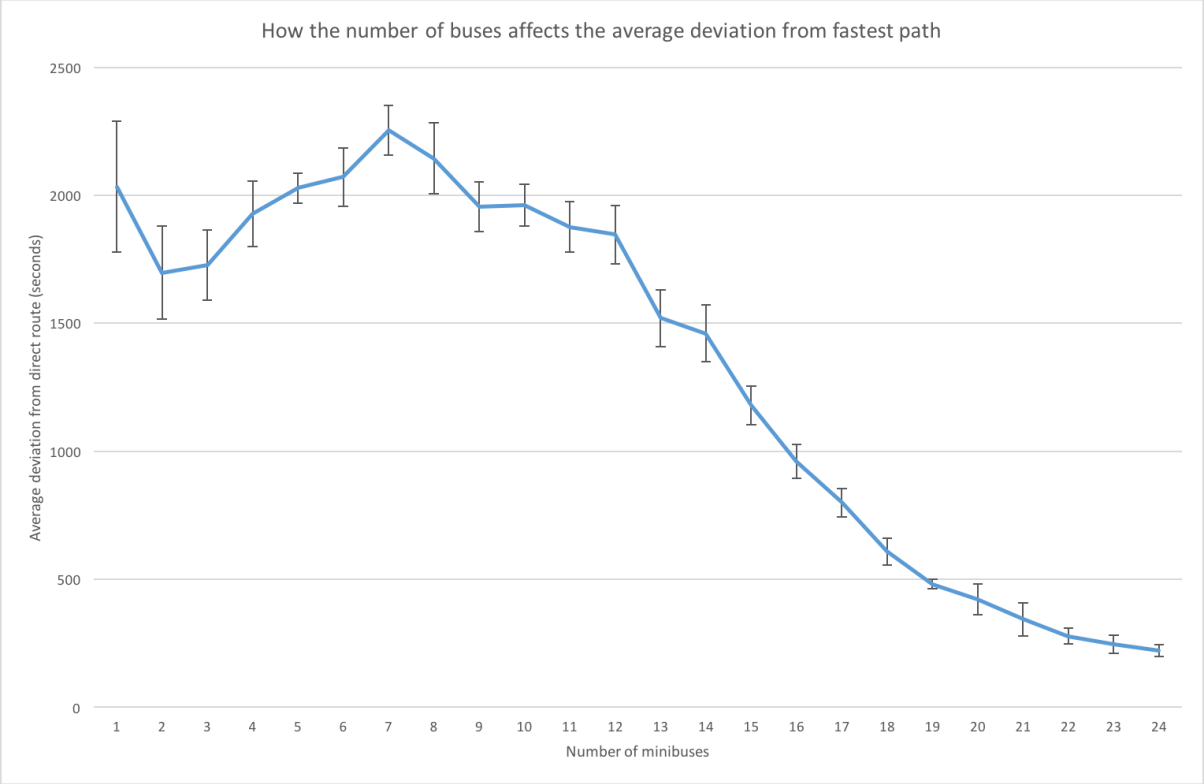
Similar to the data for average trip duration, we see that beyond 12 minibuses, the efficiency decays to an average of zero suggesting that most buses do not have any passengers. For optimum efficiency, a value of noBuses should be chosen between 7 and 10.



This data highlights that for low numbers of buses, the statistics can vary massively. This is due to the random nature of the stochastic simulator. The general trend of less waiting and less variation as buses are increased can be seen, and follows by considering that, as mentioned before, more buses mean that there will be less need to pickup a passenger while on route (the source of passenger waiting).



There is a very clear trend here. As might be expected, as the number of buses increases, the percentage of missed requests falls from around 90% to close to 0% at 16 buses. This is a good indication of how successful the public transport system is. If the missed requests percentage is too high, the system becomes a very unreliable option for people wishing to use it as their primary mode of transport. There is therefore the trade-off between lots of less-efficient buses that can always satisfy requests and fewer more-efficient buses that can't. A value of 10 here has just under 20% missed requests and is still very efficient.

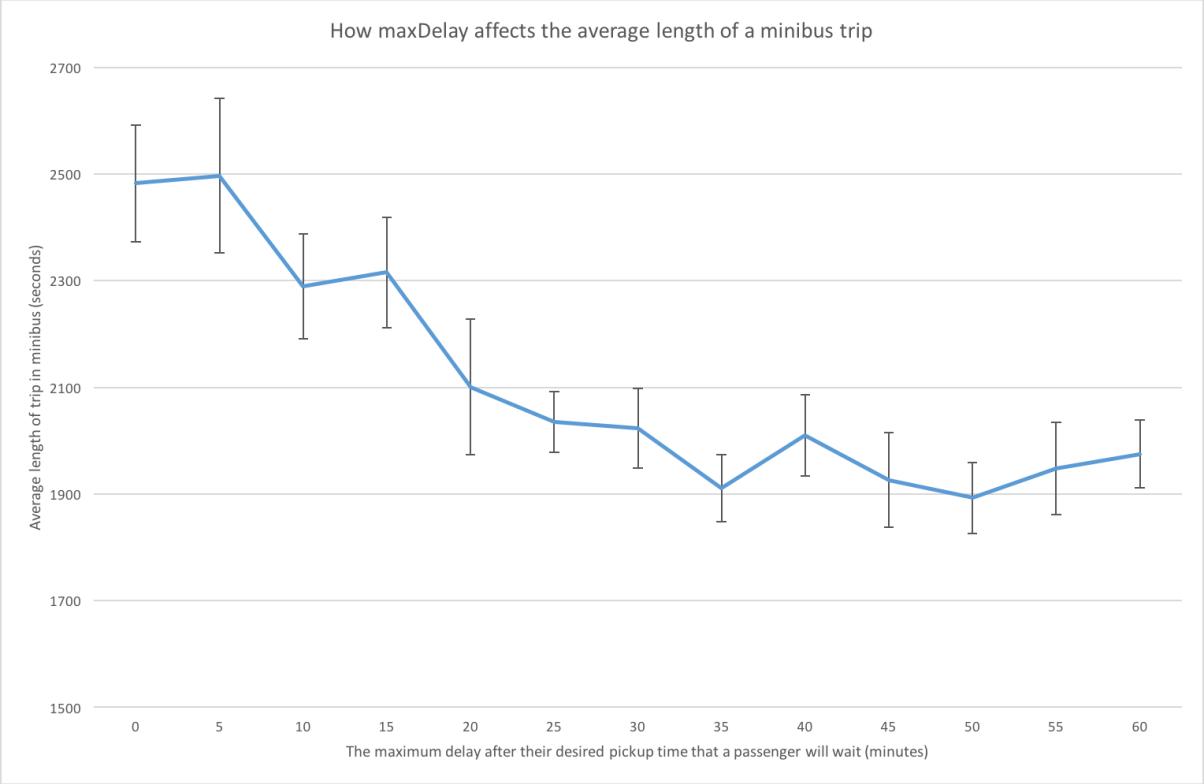


This data is further proof that beyond around 12 minibuses, the trips start to become shorter and closer to the most direct route. A value of 10 still has a high deviation from the shortest possible route of just over 32 min. This would likely be unacceptable for the majority of users and highlights a flaw somewhere in the route planning algorithm. Improvements could be made to choose a different bus if the fastest one still has passengers to drop off.

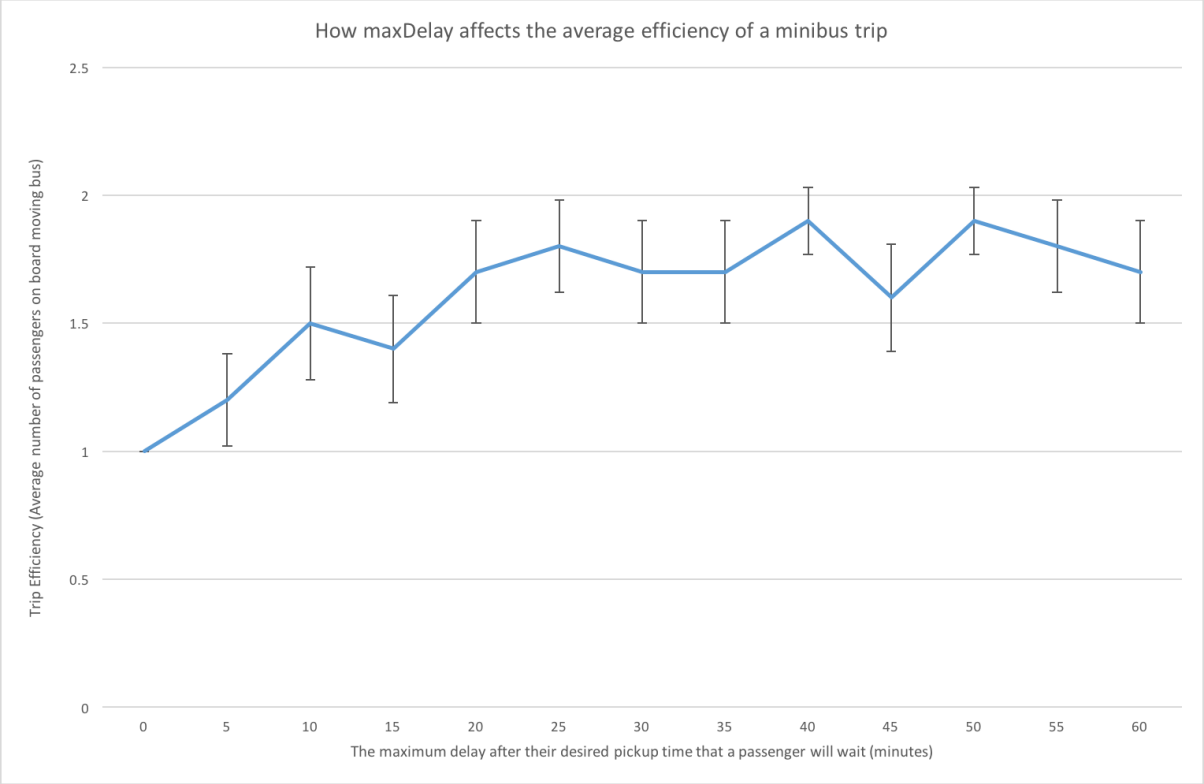
From the data collected, the only way to have a realistically usable simulator would be to have a large number of buses such as 20. At this point the system is very inefficient, most of the time buses are empty or just have a single passenger so the minibuses are essentially taxis. For a smaller number of buses such as 10, the simulator is much more efficient but potential users will have to put up with much longer journey times and possibly not even getting picked up at all.

b. Testing the effect of the maximum allowable pickup delay

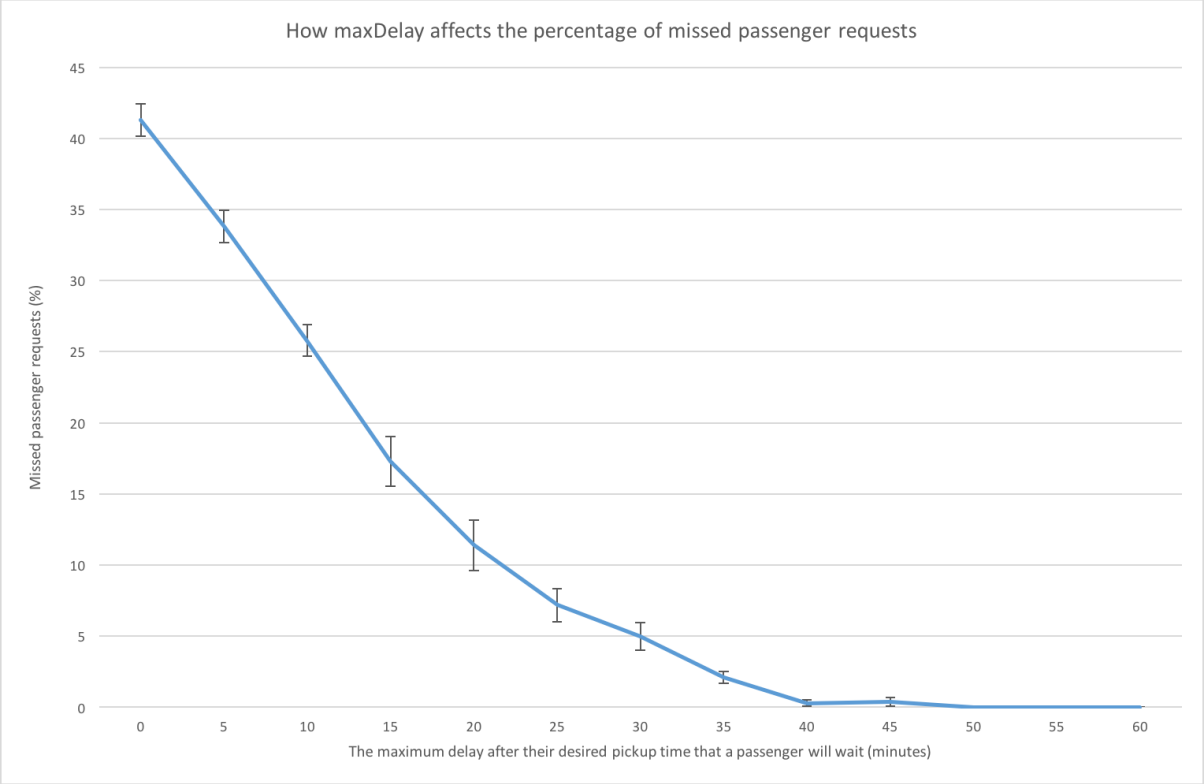
The value of 10 for noBuses was chosen from the previous set of experiments to see if it was possible to mitigate the negative effects of a more efficient simulator (long journey time, missed requests) by varying the maxDelay parameter. This controls the maximum time after a user's desired pickup time that they are happy to be picked up. I experimented in increments of 5 minutes from 0 to 60.



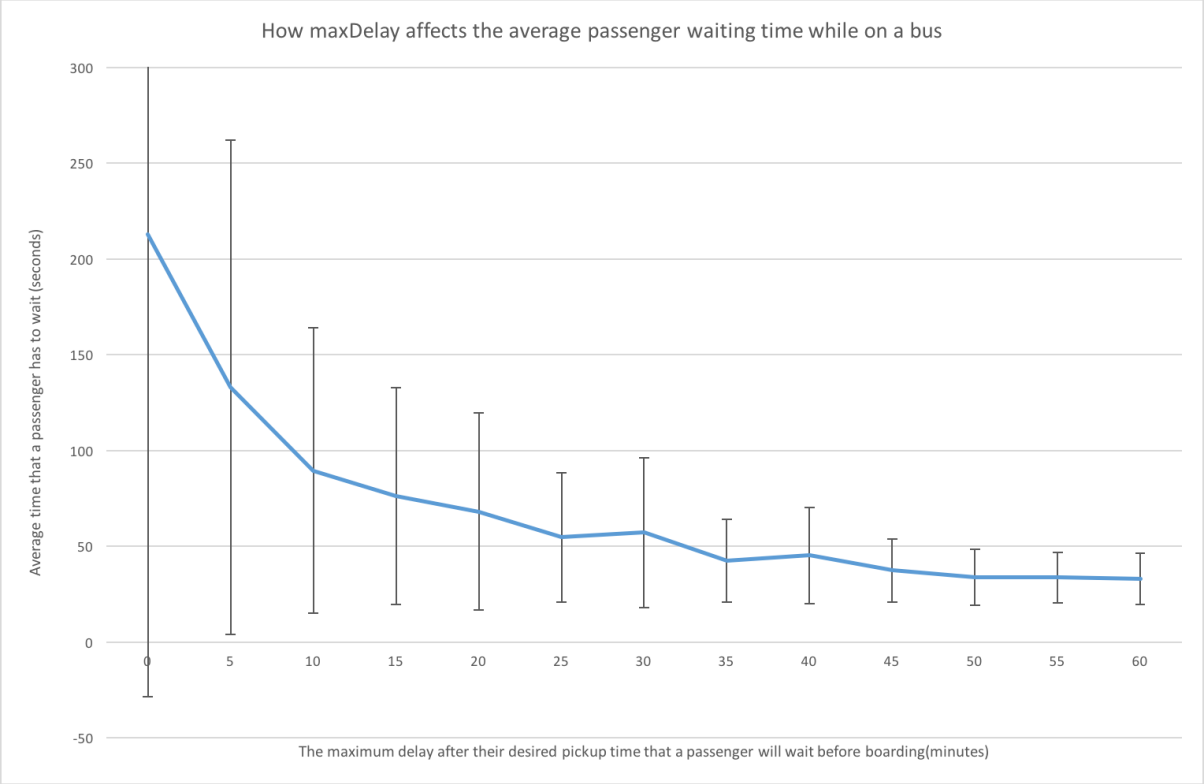
From this data we can see that the average trip length is reducing as maxDelay increases. This makes sense as the greater time window allows for more minibuses to be considered in the scheduling algorithm and thus find a faster option. However, for larger values (30 to 60 minutes) the improvement is minimal to non-existent. The optimum value for the lowest average trip length is therefore 30 minutes.



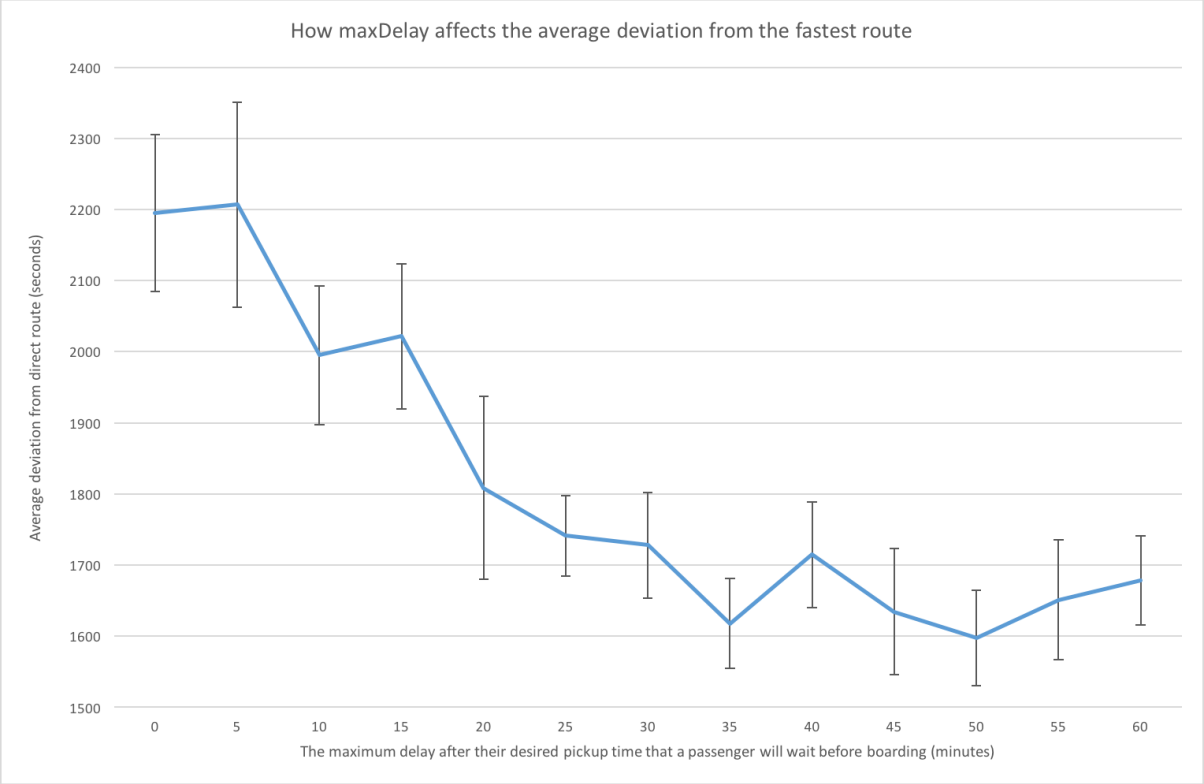
As the maxDelay increases, it can be seen that the trip efficiency also increases. Again this follows from the fact that, due to the larger window for pickups, more effective routes can be scheduled. Above around 20 minutes, the efficiency stays largely constant so the optimal value for maximising efficiency is 20 minutes.



As with varying the number of buses, increasing the maxDelay has a noticeable effect on the percentage of missed passenger requests. For values of 40 minutes and above almost no requests are missed. The same effect was achieved with 16 buses and 15 minute delays before. This indicates that increasing the maxDelay can lower the number of missed requests for a given number of buses and likewise. The optimum value here would be at 40 minutes, where no requests are missed or 30 minutes if a 5% miss percentage is allowable.



This data is almost identical to the same experiment with varying numbers of buses. There is an overall trend of lower and less variable wait times as the maxDelay increases. It is highly variable with zero or close to zero values because at that point, the buses have very small windows in which to relocate to satisfy the requests and thus there will be very few successful requests. As there are so few trips, the waiting time can vary drastically depending on the random nature of the simulation. An optimum value of 30 should be chosen as it is near the middle and any increase does not have a significant decrease in waiting time.



The average deviation from the direct route decreases as *maxDelay* increases. This makes sense as average trip duration decrease, while the shortest route stays constant. A value above 30 minutes gives a deviation of under 30 minutes which is only a small improvement over the deviation of about 32 minutes with a *maxDelay* of 15.

From these experiments I would conclude that with my simulator and input parameters as specified in the first provided example file that both an efficient and reasonable public transit system could not be implemented. For the minibuses to be even slightly efficient, the average trip ends up taking around half an hour longer than if the bus just drove straight there.

If a large number of buses are deployed then the system can be used, but the efficiency of these buses will be so low that it is likely not financially viable. If a lower number of buses is used, the maximum allowable departure delay can be increased to improve the viability of the simulator but it has less impact.

Changes to the route planning algorithm are therefore required to reduce the trip duration are needed to make the system reasonable for use (on this map). A possible source of issue is the massive delays that can occur when a bus delays at a stop, while other passengers are on board. Reworking the algorithm to avoid doing this where possible could yield massive improvements.

6. Conclusions

My implementation of the simulator works but there is much scope for improvements to the route planning and scheduling algorithms. On certain route maps, such as the one

experimented with, it appears that my algorithms cannot find an optimum solution that is both efficient and reasonable for use in real life.

These improvements should focus on the average deviation from the fastest route to be minimized while keeping the efficiency high and preventing buses from waiting at stops while they have passengers on board.