Computer Science Large Practical
2015-2016

Report

Written by s1237357

December 20, 2015

Introduction

The requirement for the Computer Science Large Practical is to develop a command-line application in the C programming language. The purpose of the application is to execute stochastic simulations of an on-demand public transport system for future cities, in order to gain insights into how the efficiency of this operation and the customers' satisfaction can be optimized.

Requirements

• Passengers only board or disembark at designated minibus stops. Once a request enters the system, it has to be queued in view of scheduling.
• Users place journey requests at exponentially distributed time intervals with the mean delay given as an input parameter.
• A request comprises randomly chosen departure and destination stops, a desired boarding time that is an exponentially distributed delay after the request time, whose mean is also given as an input parameter.
• A journey request may be served within a maximum admissible delay after the desired departure time, which is be given as an input parameter.
• The route of a minibus that is already in service can be updated to accommodate a new request as long as this will not alter previously agreed departure times.
• The minibus occupancy cannot be exceeded at any time.
• Each event is outputted on a single line in the following format: <**time**> **->** <**event**> <**details**> where time is in <**days:hours:minutes:seconds**> format.

Application should return summary statistics at the end of the simulation time:
• Average duration of trips performed by a minibus over the duration of the simulation.
• The trip efficiency as the average number of passengers transported per unit of travel time.
• Percentage of missed requests as fraction of missed over total requests
• Average passenger waiting time (in seconds) as average duration waiting at stops during a trip (except for boarding and disembarkation of others).
• Average trip deviation as sum of differences between shortest and actual trips over trip count.

• Simulator should allow experimenting with different values of the maximum admissible pick-up delay and the number of minibuses. For that, in the input file it should be able to specify a set of values instead of a single one. This set is defined by using the **experiment** keyword, followed by the values. Also, it should allow experimenting with multiple parameters at the same time.

Program Design

The application can be run by following the instructions that are given in the README file.
In order to design the simulator that meets all requirements above, the following design (Figure 1.) is chosen. Diagram below represents the skeleton of the program and how modules are connected to each other by using header files as well as what important functions are used in each module.

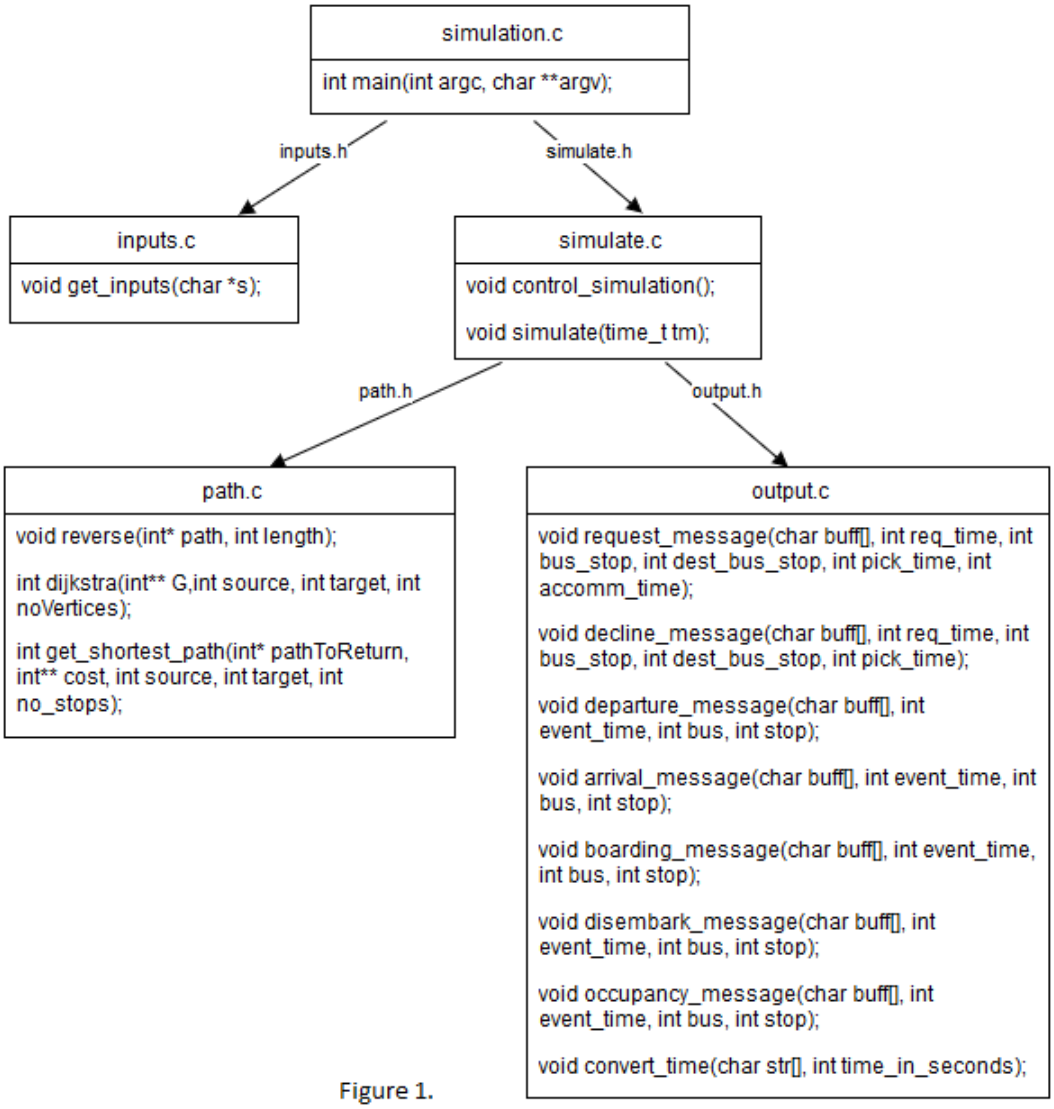

Figure 1.

Detailed Description of Modules

Module simulation.c

This module contains main function `int main(int argc, char **argv)` that takes command line parameter as an argument. Also in `main()` there are two function calls:

- `get_inputs(argv[1])` that passes command line argument as the name of input file from which parameter values will be retrieved.
- `control_simulation()` that after retrieving inputs starts main simulation control.

Module inputs.c

In this module `get_inputs(char *s)` function is defined. The function checks if input file name is valid, if it is no valid, error will be raised and program will exit. Variable values from input file are retrieved by using 'while' loop which is executed until all lines of document were scanned. Every line is scanned using C library function `fgets(char *str, int n, FILE *stream)`. The line then is saved intotemprary buffer and tokenised using another library function strtok(char *str, const char *delim). Delimiter string is set to " $\backslash n \backslash t$" so that it is sensitive to white space, tab, and new line. `If` statements check what keyword is currently being parsed, and then retrieves its corresponding value. If keyword is **map**, then next file line is scanned and first line of the map is saved into `temp` array with maximum size of 65,535. This will allow us to know of what size 2-dimensional array we need to allocate memory and therefore the first line of `temp` is copied to a new matrix **map**. Then we also

know how many lines of the file we need to scan more and we write retrieved values directly into **map**. In addition, we check if there is **experiment** keyword. If this keyword occurs, then corresponding array of maximum 100 elements, that will save experiment values, is allocated. Also, corresponding was_allocated[] element is set to 1 so that later it would be known which experiment arrays needs to be freed and which values will be used for experimentation. In the end of the module corresponding rotation values are calculated to each keyword. During the experimentation this will allow perform experiment with all possible combinations of given experiment values.
Function `get_inputs()` also uses helper functions `myStrToInt()`, `myStrToUnsigned()`, and `myStrToFloat()` that converts strings into integer, unsigned integer, and float numbers respectively. If it fails to do it, error is raised and program exits.
In addition, the following external variables are defined so that it would be reachable for **simulate.c** module:

extern unsigned int busCapacity
extern float requestRate
extern unsigned int maxDelay
extern unsigned int noStops
extern float stopTime
extern int *busCapacityExp
extern int *boardingTimeExp
extern float *requestRateExp
extern float *pickupIntervalExp
extern int *maxDelayExp
extern int *noBusesExp
extern float *stopTimeExp
extern int was_allocated[7]
extern unsigned int boardingTime
extern float pickupInterval
extern unsigned int noBuses
extern int **map
extern int numberOfExperiments
extern int busCapacityExpLength
extern int boardingTimeExpLength
extern int requestRateExpLength
extern int pickupIntervalExpLength
extern int maxDelayExpLength
extern int noBusesExpLength
extern int stopTimeExpLength

Variables below are rotation intervals for corresponding experiment values (after how many experiments we need to change each value so that experimentation would be performed over all possible combinations):
extern int capacityRot, boardTimeRot, requestRot, pickupRot, delayRot, busesRot, stopsRot, stopTimeRot.

Module path.c

This module has three functions:

1. Function `dijkstra(int** G,int source, int target, int noVertices)` takes cost matrix, source and target vertices, and size of cost matrix as arguments and returns shortest distance between source and target vertices. This is implementation of shortest distance Dijkstra algorithm.

2. Function `get_shortest_path(int* pathToReturn, int** cost, int source, int target, int no_stops)` takes an array `pathToReturn` as an argument. Into this array resulting path will be written. Also it takes cost matrix, source and target vertices, and size of cost matrix as arguments. In this function most of the code is copied from `dijkstra()` function because these two functions are used for different purposes and usually performed by using different arguments. Therefore, they were separated in order to avoid overriding of `pathToReturn` argument. Also this function returns length of the shortest path (number of vertices that it goes through) while `dijkstra()` returns shortest distance.

3. Function `reverse(int* path, int length)` reverses path that was found by using function `get_shortest_path()`. This needs to be done because `get_shortest_path()` finds path by backtracking from an array that holds predecessor for every visited vertex, therefore algorithm backtracks from target to the source vertex.

Module output.c

All functions below format and write corresponding message into to passed argument `buff[]`:
`void request_message(char buff[], int req_time, int bus_stop, int dest_bus_stop, int pick_time, int accomm_time)`
`void decline_message(char buff[], int req_time, int bus_stop, int dest_bus_stop, int pick_time)`
`void departure_message(char buff[], int event_time, int bus, int stop)`
`void arrival_message(char buff[], int event_time, int bus, int stop)`
`void boarding_message(char buff[], int event_time, int bus, int stop)`
`disembark_message(char buff[], int event_time, int bus, int stop)`
`void occupancy_message(char buff[], int event_time, int bus, int stop)`
Passing argument and writing answer into it rather then just returning it, helps to avoid additional memory allocation. Function `convert_time(char str[], int time_in_seconds)` converts given time argument in seconds to appropriate output time format <**dd:hh:mm:ss**> and writes it into passed argument `str[]`.

Module simulate.c

In this module data structure Bus is defined:
struct Bus {
number of passengers currently on board
int passengersOnBoard;
if bus arranges to pick up a apassenger, seat is reserved for him
int reservedSeats[6500];
bus stops from which bus picking up passengers (planned route)
int route[6500];
number of stops that bus will visit
int routeLength;
bus stop at which bus currently is
int currentStop;
the bus stop that is last to visit
int lastScheduledStopIndex;
leaving time (in seconds) to a respective bus stop in a route
int departureTime[6500];
leaving time (in seconds) to a respective bus stop in a route
int arrivalTime[6500];
arrived tracks whether bus arrived to the bus stop to pick or disembark passenger
int arrived;
tripDuration[] tracks actual trip duration to the bus stop parralel in the bus route
int tripDuration[6500];
theoreticalDuration[] tracks shortest trip duration to the bus stop parralel in the bus route
int theoreticalDuration[6500];
destination bus stops that every passanger goes to
int passengersToDisembark[6500];
waiting[] tracks for how many passengers bus will be waiting at the bus stop parallel to
the stops int the bus route
int waiting[6500];
tracks travel of the whole route (only when bus is moving - waiting, boarding and disembarking at the stop excluded)
int routeDuration;
tracks the total number of passengers transported between each
stop in the bus route (when bus is moving)
int passengersTransported;
};

When calling function `simulate()`, an array of structures is defined and initialized. The length of array is equal to the number of buses thus every bus is has its own data. This module has two main and most important functions that controls simulation behaviour:

1. Function `control_simulation()` checks whether it is experimentation or only one-off simualtion. If it is experimentation, then it rotates the arrays of all experimentation variables with their corresponding rotation interval and prints off experimentation number and corresponding values of its parameters. Also it disables detailed output.

2. Function `simulate()` has functions defined inside of it that helps to control buses:

• `is_valid_subroute()` checks if path from pickup stop to destination stop is subroute of current given bus route (returns index of bus route array at which subroute begins, if path is not a subroute, -1 is returned).

• `add_route()` merges path and current bus route if merge is possible and updates information about how many passengers will be boarded/disembarked and at which bus stops.

• `refresh_bus_timing()` refreshes bus timing for a given bus (updates departure time if more passengers will be boarded and updates further arrival times as it will be delayed).

• `scan_capacity()` checks what will be bus capacity at the last scheduled stop (latest bus stop from which passenger will be picked up). This function is useful when checking if it is possible to schedule passenger pickup without violating maximum bus capacity constraint.

• `accommodate_passenger()` function checks whether is it possible to pickup customer. At first it tries to pick passenger on its way to some other destination bus stop, then it checks whether passenger needs to picked up from the bus stop to which it is going, and lastly it tries to schedule picking up by making the bus to wait at the bus stop. If none of these conditions are possible then bus can't pick the customer.

• Functions `disembark()`, `board()`, and `shift()` updates bus data respectively by reducing/increasing number of passengers that needs to be boarded/disembarked and removes unnecessary data by shifting it.

• `find_next_event()` checks whether there is any bus that has to perform an event that time is less than the current time. This function contains **while(1)** loop which is the main event controller. Loop can only be broken if current time exceeds specified simulation limit. Logic in the loop function is simple:

1. Checks if there is any bus that needs to perform the event with time less than current time until no such buses are found

2. If such bus was found, check what event needs to be performed: arrival/departure/boarding/disembarking or it only passes through the stop and data needs to be shifted - trigger corresponding message and search again for the next event.

3. If no more events are found, then check for the newest request whetheher exists any bus that could accept it and corresponding message is triggered.

4. If request was accepted, then bus timing is scheduled and data for corresponding bus is updated.

5. Repeat all 3 steps above until simulation time reaches its limit

Parser Testing

In order to test if parser works as required and it raises warnings or errors if any invalid or mistyped inputs occur, corresponding test were performed.

• `invalid_input_test1` was used as input file with multiple errors: used double where unsigned integer or integer was required, added letter next to number. Program raised errors as expected:

```
Could not convert to integer "12.5". Input file line: 5
Could not convert to integer "10s". Input file line: 6
Could not convert to integer ".9". Input file line: 17
Program exited
```

Figure 2.

• `invalid_input_test2` was used as input file with multiple errors: number of stops does not match with given map matrix dimension, double used where integer is required. Program raised errors as expected:

```
Could not convert to integer ".9". Input file line: 16
Number of bus stops does not match with number of columns and rows in the map matrix
Program exited
```

Figure 3.

• `invalid_input_test3` was used as input file with incorrect `noStops` keyword name. Program raised error as expected:

```
Number of bus stops does not match with number of columns and rows in the map matrix or wrong keyword
Program exited
```

Figure 4.

• `invalid_input_test4` was used as input file with incorrect `boardingTime`. As expected, program raised warning but performed simulation (boardingTime was set to default 0):

```
Warning: boardingTime was given as 0 or incorrect keyword
00:00:00:22 -> new request placed at stop 3 to stop 2 for departure at 00:00:29:44 cannot be accommodated
00:00:05:05 -> new request placed at stop 4 to stop 2 for departure at 00:00:26:48 cannot be accommodated
00:00:07:14 -> new request placed at stop 5 to stop 1 for departure at 00:00:45:23 cannot be accommodated
00:00:08:27 -> new request placed at stop 2 to stop 5 for departure at 00:01:54:40 cannot be accommodated
00:00:18:42 -> new request placed at stop 2 to stop 5 for departure at 00:01:30:57 cannot be accommodated
00:00:25:00 -> new request placed at stop 1 to stop 5 for departure at 00:00:35:18 cannot be accommodated

---
average trip duration 00:00
trip efficiency 0.000000
percentage of missed requests 100.00
average passenger waiting time 0 seconds
average trip deviation 0.00
---
```

Figure 5.

• `experiment_noBuses` was used as input file with `noBuses experiment` keywords and corresponding different values. Program executed as expected: outputted summary statistics after every experiment, detailed simulation output was disabled.

Program Testing and Debugging

For program debugging I have used **GNU GDB** debugger. When program used to crash it helped me to backtrack what went wrong.
For program testing I have chosen to write bus data into separate file after every event. For example: **Figure 6.** represents event sequence after checking if there are any events that occurs before current time. In line 348 next event bus number is -1 which indicates that there are no such buses that has event earlier than current time. Lines 349-350 represents that new request enters the system and it is being searched for a bus that could accommodate this request. It is printed which bus data is currently being checked, also current time and request times are displayed as well as pickup and destination bus stops. In line 350 agreement method is printed: accommodated by extension means that bus will disembark passenger at stop 3 and from there will go straight to pickup passenger that sent this request.
Lines 353-360 represents most important data about the bus:
• `ROUTE` represents already scheduled route
• `RESERVED` represents how many passengers will be boarding at each bus stop
• `DISEMBARK` represents how many passengers will be disembarking at each stop
• `ARRIVAL` represents arrival time to each bus stop

• `DEPARTURE` represents bus departure time from each bus stop
• `WAITING` represents for how many passengers bus will be waiting at corresponding bus stop
• `ACTUAL DURATION` represents how long in total does it take for all passengers to get to the corresponding bus stop
• `SHORTEST DURATION` represents shortest duration for all passengers in total that going to corresponding bus stop
Lines 353-360 and 362-369 respectively represents bus data before and after recalculating its timing.

```
348  NEXT EVENT BUS: #-1
349  ------ACCOMMODATING----BUS #2  ----CURRENT TIME: 946  --- PICK TIME: 2231  --FROM: 5 -- TO: 1-----
350      Accommodated by extension
351
352  BEFORE REFRESHING TIMING----------------------------------------
353  ROUTE: [5][4][1][2][1][2][3][5][4][1]
354  RESERVED:     [3][0][0][0][0][0][0][1][0][0]
355  DISEMBARK:    [0][1][1][1][0][0][1][0][0][1]
356  ARRIVAL:      [1400][1670][1740][2050][2490][2800][2920][3050][3300][3360]
357  DEPARTURE:    [1430][1680][1750][2190][2500][2800][2930][3060][3300][3370]
358  WAITING:    [0][0][0][0][1][0][0][0][0][0]
359  ACTUAL DURATION: [0][260][320][620][0][0][420][0][0][0]
360  SHORTEST DURATION:  [0][240][300][600][0][0][420][0][0][0]
361  ----------------------------------------------------
362  ROUTE: [5][4][1][2][1][2][3][5][4][1]
363  RESERVED:     [3][0][0][0][0][0][0][1][0][0]
364  DISEMBARK:    [0][1][1][1][0][0][1][0][0][1]
365  ARRIVAL:      [1400][1670][1740][2050][2490][2800][2920][3050][3300][3360]
366  DEPARTURE:    [1430][1680][1750][2190][2500][2800][2930][3060][3300][3370]
367  WAITING:    [0][0][0][0][1][0][0][0][0][0]
368  ACTUAL DURATION: [0][260][320][620][0][0][420][0][0][300]
369  SHORTEST DURATION:  [0][240][300][600][0][0][420][0][0][300]
```

Figure 6.

Figure 7. represents that there was found a bus that has an event earlier than the current time. Also, the information about current time and bus last scheduled pickup stop is displayed. Main bus information is printed out as well as the message of the event (lines 379, 389, and 399).
From bus data in the lines 373-378 we can see that this arrival event since bus needs to pick a customer from bus stop 2 and arrival time to this stop (992 seconds) is less than current time (1126 seconds). Thus arrival message is triggered. When analyzing behaviour of such simulation, I have discovered that simulation used to board/disembark customers before next request message was printed, for example:
00:00:47:42 -> minibus 3 disembarked passenger at stop 1
00:00:47:42 -> minibus 3 occupancy became 0
00:00:47:52 -> minibus 3 boarded passenger at stop 1
00:00:47:52 -> minibus 3 occupancy became 1
00:00:47:47 -> new request placed from stop 2 to stop 3 for departure at 00:01:05:32 scheduled for 00:01:05:32
00:00:48:02 -> minibus 3 boarded passenger at stop 1
Obviously, request message should have been printed before minibus 3 boarded passenger. Therefore, when bus arrives to a bus stop, we know that there is a passenger to disembark or to board. Thus, after bus arrives and arrival message is yield, arrival time is increased by boarding time immediately because next time arrival time is used for boarding/disembarking time. The same is done after disembarking passenger: if there are more passengers to board or disembark, arrival time is increased again. After boarding time is increased only if there are more passengers to board. After bus leaves (line 399), previously stored bus data will not be needed anymore, therefore data is removed/shifted (line 401).

```
NEXT EVENT BUS: #3
 ---------- NEXT EVENT ---------
CURRENT TIME: 1126 LAST SHED. STOP INDEX 4
BUS #3 ROUTE: [2][3][4][1][5][4][1][2]
RESERVED:     [1][0][0][0][1][0][0][0]
DISEMBARK:    [0][0][0][1][0][0][0][1]
ARRIVAL:      [992][1122][1242][1302][1732][1982][2042][2342]
DEPARTURE:    [1002][1122][1242][1312][1742][1982][2042][2352]
WAITING:    [0][0][0][0][0][0][0][0]
-----------------------------------BUS #3 ARRIVED TO THE STOP #2
NEXT EVENT BUS: #3
 ---------- NEXT EVENT ---------
CURRENT TIME: 1126 LAST SHED. STOP INDEX 4
BUS #3 ROUTE: [2][3][4][1][5][4][1][2]
RESERVED:     [1][0][0][0][1][0][0][0]
DISEMBARK:    [0][0][0][1][0][0][0][1]
ARRIVAL:      [1002][1122][1242][1302][1732][1982][2042][2342]
DEPARTURE:    [1002][1122][1242][1312][1742][1982][2042][2352]
WAITING:    [0][0][0][0][0][0][0][0]
---------------------------------------BUS #3 BOARDED PASSENGER AT THE STOP #2
NEXT EVENT BUS: #3
 ---------- NEXT EVENT ---------
CURRENT TIME: 1126 LAST SHED. STOP INDEX 4
BUS #3 ROUTE: [2][3][4][1][5][4][1][2]
RESERVED:     [0][0][0][0][1][0][0][0]
DISEMBARK:    [0][0][0][1][0][0][0][1]
ARRIVAL:      [1002][1122][1242][1302][1732][1982][2042][2342]
DEPARTURE:    [1002][1122][1242][1312][1742][1982][2042][2352]
WAITING:    [0][0][0][0][0][0][0][0]
---------------------------------------BUS #3 LEFT STOP #2
--------------------------------------------------------------------------------
BUS #3 SHIFTING---
```

Figure 7.

Memory Management

In order to check any possible memory leak, I have used **Valgrind**. virtual machine.
Figure 8. represents heap summary after running one-off simulation with `simple_input` file.

```
==21058== HEAP SUMMARY:
==21058==     in use at exit: 0 bytes in 0 blocks
==21058==   total heap usage: 72,649 allocs, 72,649 frees, 6,759,855 bytes allocated
==21058==
==21058== All heap blocks were freed -- no leaks are possible
==21058==
==21058== For counts of detected and suppressed errors, rerun with: -v
==21058== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Figure 8.

Figure 9. represents heap summary after running the experimentation with different parameter `noBuses` values from file `simple_experiment`.

```
==21210== HEAP SUMMARY:
==21210==     in use at exit: 0 bytes in 0 blocks
==21210==   total heap usage: 1,207,510 allocs, 1,207,510 frees, 108,459,419 bytes allocated
==21210==
==21210== All heap blocks were freed -- no leaks are possible
==21210==
==21210== For counts of detected and suppressed errors, rerun with: -v
==21210== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Figure 9.

Figure 10. represents heaps summary after running program with invalid input script `test_invalid_input1`.

```
==21275== HEAP SUMMARY:
==21275==     in use at exit: 0 bytes in 0 blocks
==21275==   total heap usage: 3 allocs, 3 frees, 1,704 bytes allocated
==21275==
==21275== All heap blocks were freed -- no leaks are possible
==21275==
==21275== For counts of detected and suppressed errors, rerun with: -v
==21275== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Figure 10.

Purpose of all these checks is to guarantee that all memory allocated variables were freed after program termination or exit after error was caught.

Shortest Distance/Path Algorithm

In order to find shortest distance or shortest path when required, I have chosen to use Johnson's algorithm. Other algorithm candidates were Dijkstra's and Bellman-Ford algorithms. However, map input adjacency matrix is given with negative edges. Therefore, Johnson's algorithm allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the BellmanFord algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph. Worst case time complexity of this algorithm is: $O(EV + V^2 \log V)$

Statistics Analysis

For output summary visualization I have used `GnuPlot` software. I have run various experimentations to check and analyze simulator's efficiency. Initially using input file `grid` the following experiments were performed:

• **busCapacity experiment 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15**

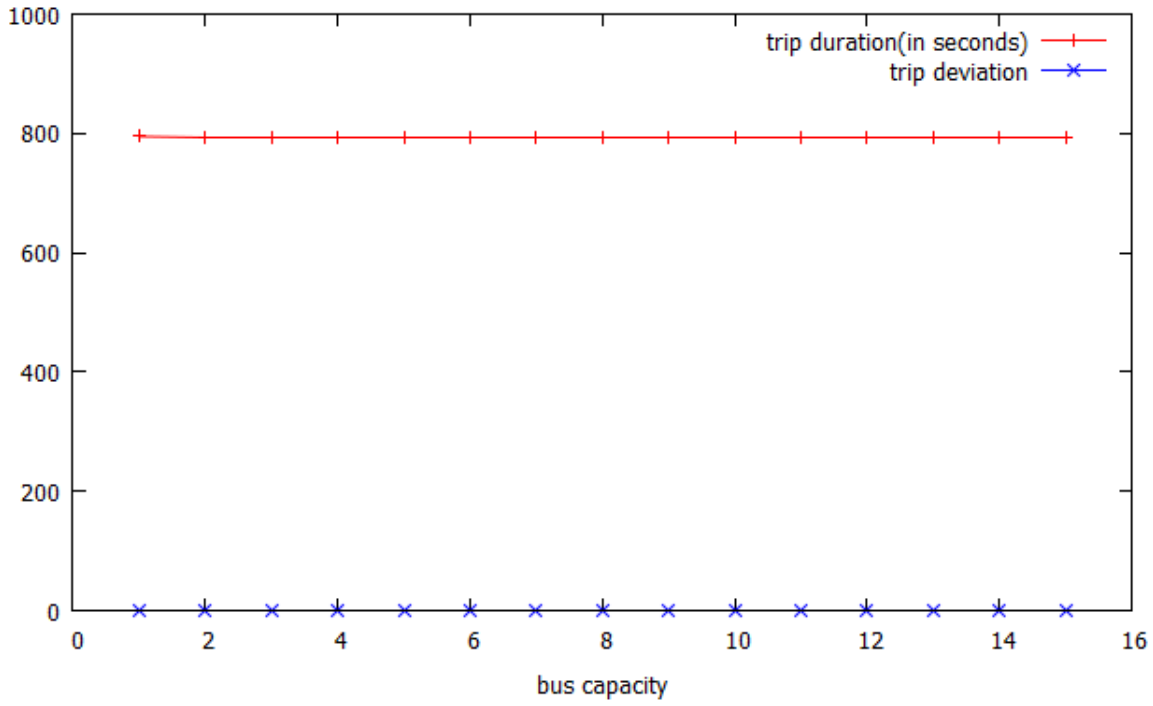

Figure 11.

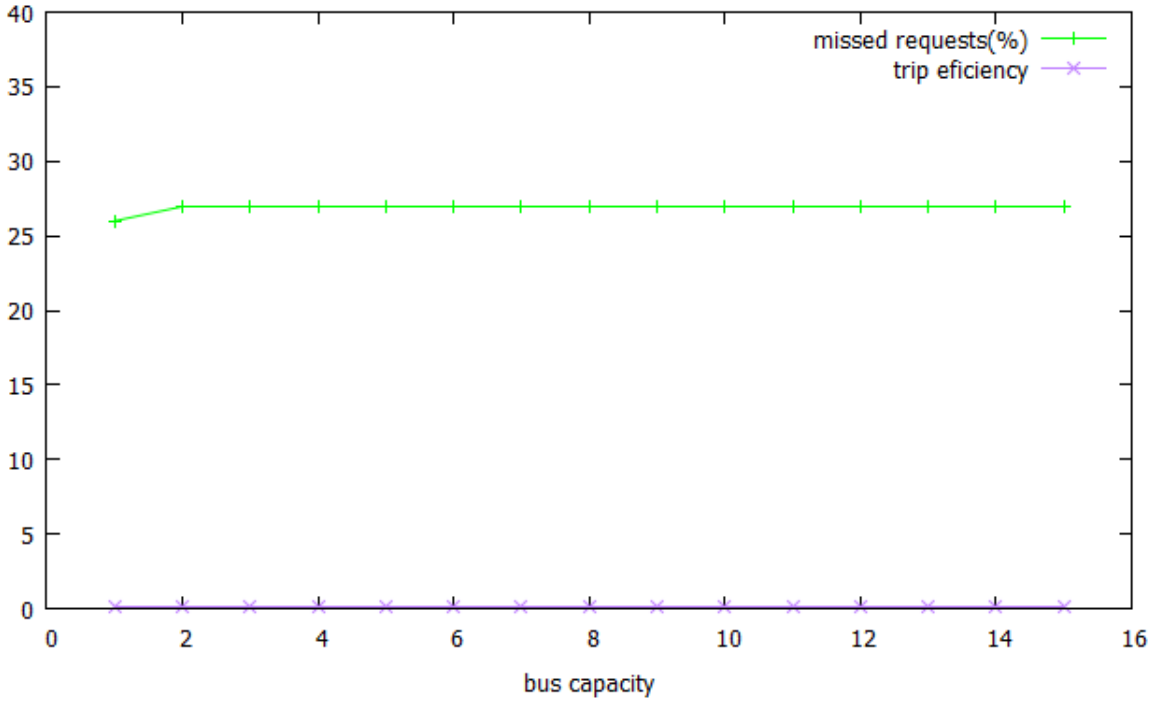

Figure 12.

• **boardingTime experiment 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20**

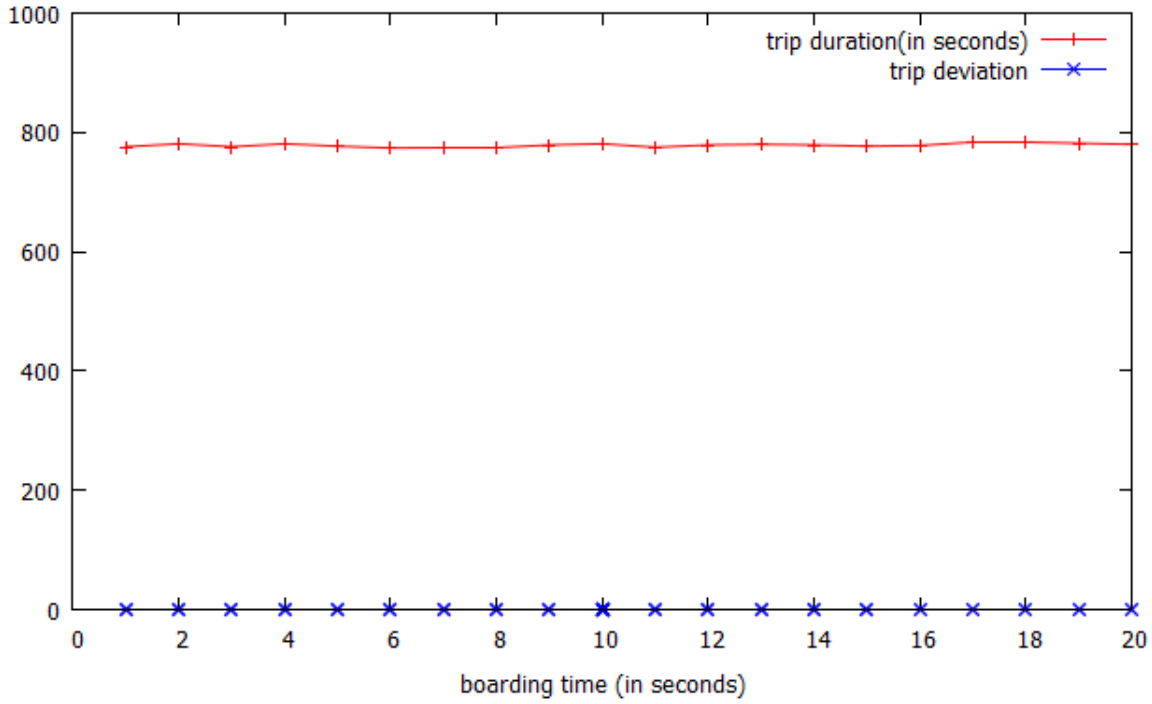

Figure 13.

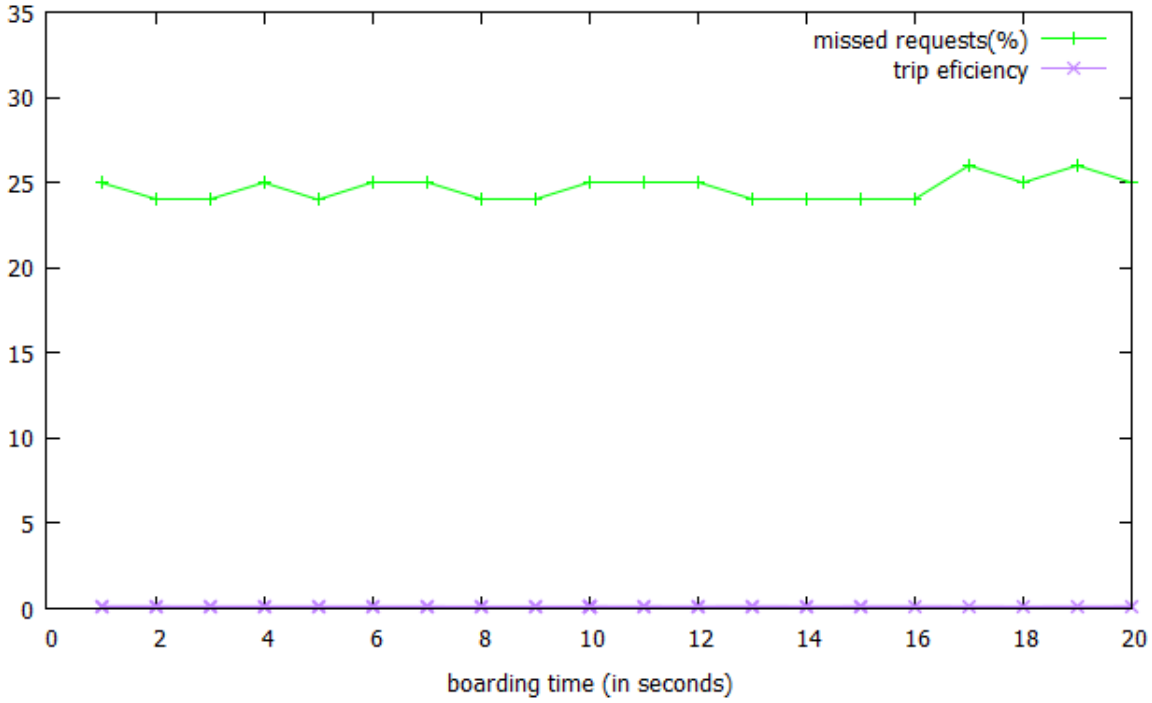

Figure 14.

- **maxDelay experiment 5 10 15 20 25 30**

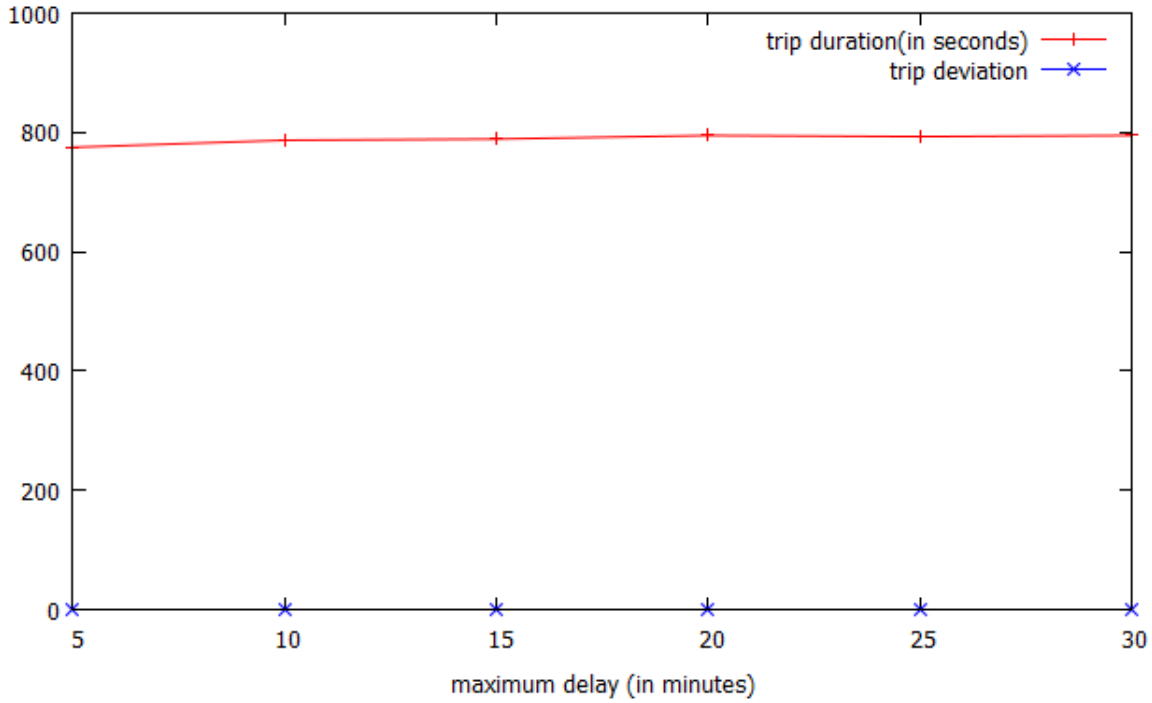

Figure 15.

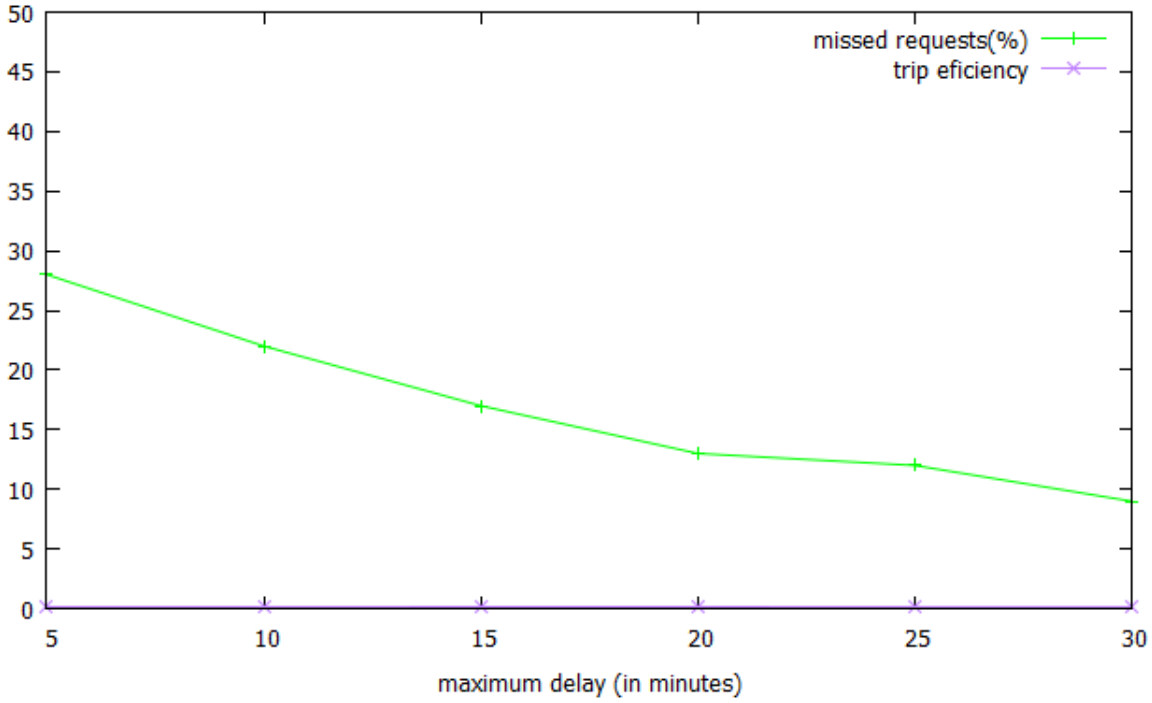

Figure 16.

• **noBuses experiment 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30**

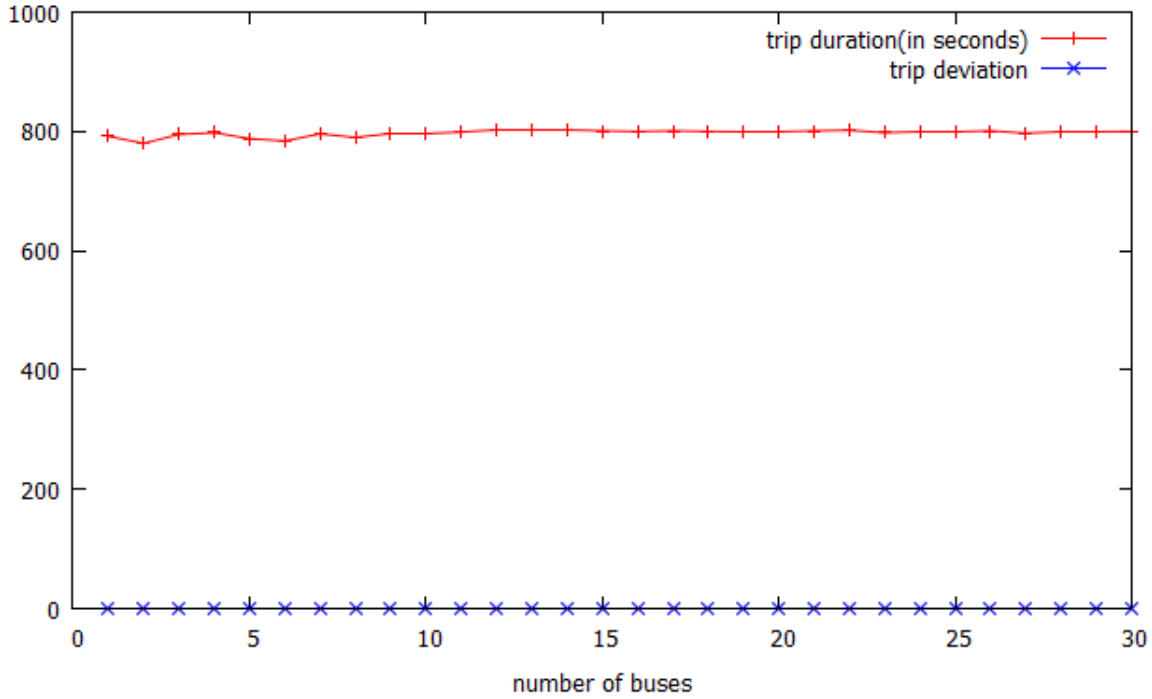

Figure 17.

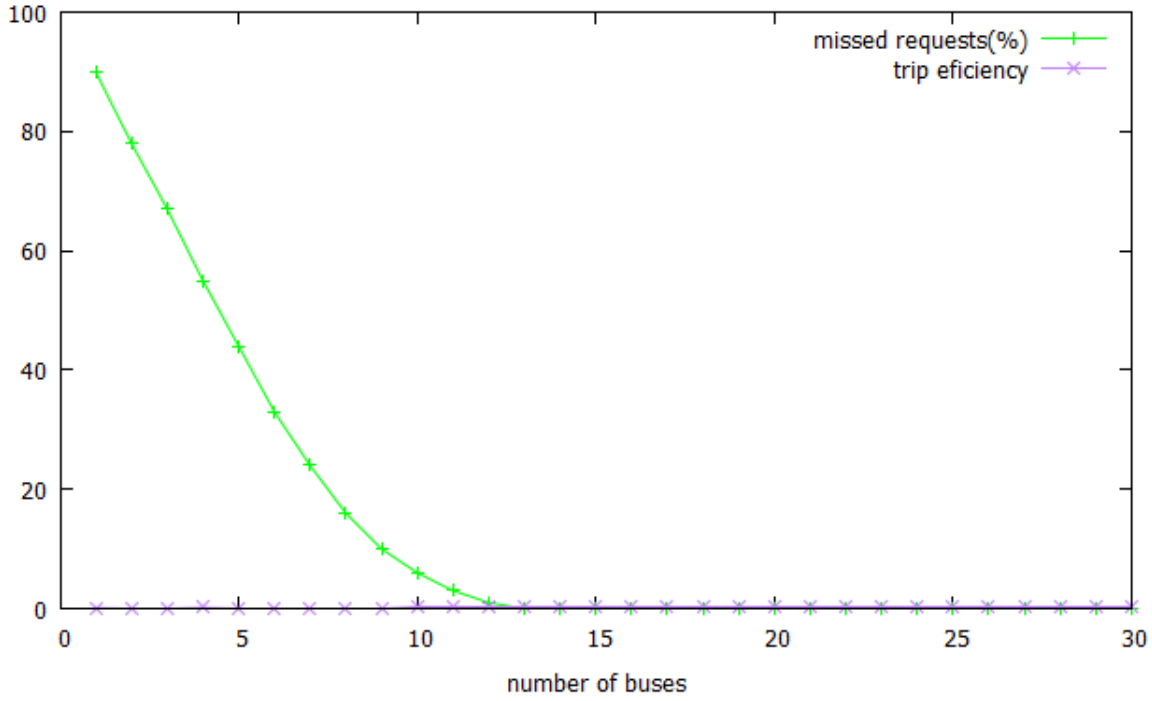

Figure 18.

From the graphs above we can see that only number of buses and maximum time that user can wait beyond the desired pickup time influences percentage of missed requests. Greater delay time and bus number lowers number of missed requests. All other simulation efficiency parameters are not affected. Also, I did not include average passenger waiting time, because it will never be greater than 0. This is obvious disadvantage of my program design and could be improved in future implementation. This happens because my program schedules following pickup time only after the time of last passenger pickup, e.g. if we know that bus will be waiting at bus stop #1 for a passenger and if we know when bus arrives and leaves that stop, and also if there are no more pickups scheduled, program does not try to pickup another passenger from stop #1 if he wants to board while bus is standing at the stop #1. This disadvantage also affects trip efficiency (number of passengers transported per unit time).

Conclusion

My simulation program will always have low trip efficiency, on the other hand it has low trip deviation, which means that bus always takes the shortest path and transports passengers at its quickest, however this also means, that the bus does not pickup a lot of passengers on its way from pickup to destination stop. Furthermore, my simulation on average cannot accommodate 20-25% requests which can be reduced even more by increasing number of buses and maximum delay time. Not surprisingly, to define what is efficiency of such simulation, is NP-hard problem.