# REPORT
# - Simulator

Computer Science Large Practical

Ludwik Bacmaga (s1345559)

## Brief
In this project we were meant to produce a automated simulator on a transport system based on a given input, which will produce different bus journey which will pick up as many customers as possible and still be as efficient as possible with time. Our inputs were provided from the separate file, which then we had to read and use the given information in the input file in the rest of the simulator.

## Stages of Simulator
In this simulator, firstly the file is being read into the program itself, and it checks if the program even exists or not, this is done by an ***"if"*** statement and if the file is not present in the given directory then it provides an error message and exits program.

```
if ((file = fopen(argv[1], "r+")) == NULL) {
       printf("Error! opening file");
       exit(1); // Program exits if file pointer returns NULL. /
    }
```

If there are no problems, and the input file opens, it gets parsed, and the given values that are stored in the input file get saved in the variables in the program, which before has been globally initialised, so they could be used throughout the program and no only locally. I did the same thing with time, separating day, hour, minute, seconds as the global variables.

//variables
```
struct MyInput
{
        unsigned int busCapacity;
        unsigned int boardingTime;
        float requestRate;
        float pickupInterval;
        unsigned int maxDelay;
        unsigned int noBuses;
        unsigned int noStops;
        int **map;
        float stopTime;
};

struct MyInput input;
```

In this code, "int **map;" means that the "map" is a double array, which in the parsing stage I allocate memory to it to be the size of "noStops" for the rows and columns. I do this using the malloc function. This will save the whole map of the stops, and create a directed graph between bus stops, allowing later to work out the time (in minutes) to transport from one bus stop to another and the time taken is computed using algorithm.
After the parsing is complete, and there are no errors, the requests, times, stops, distances are calculated. For this simulator I used Dijkstra's Algorithm, as I found it easy to understand and to implement (more in Design Choices). This provided me with the time it takes to move between stops and allowed the whole simulator to work.
The requests are being created using random function (exponentially distributed) which will determine when the new request is made.
After the new request was made, there were outputs depending on what the simulator did (if the bus was leaving bus stop, arriving bus stop, passenger went in or out of bus, or if there is a new request).
It would carry on in a loop until the final time (given in the input file) was in met in the simulation (for example 24 hour), the simulation would finish and the analysis of the

program would be printed to show how efficient the simulation was,

## Design Choices

First design choice was to create the variables from input file, and the time as global variables, in a struck group (input as one struck and time as another). This allowed me to distinguish and allow easier access to the variables when I needed them, anywhere in the program, and I would know where they belong because they would have the struck name at front (for example time.day would have “time.” at the front which let’s me know that it is the global variable from time).

```
struct MyInput
{
        unsigned int busCapacity;
        unsigned int boardingTime;
        float requestRate;
        float pickupInterval;
        unsigned int maxDelay;
        unsigned int noBuses;
        unsigned int noStops;
        int **map;
        float stopTime;
};

struct MyInput input;

struct Time
{
        int day;
        int hour;
        int min;
        int sec;
};

struct Time time;
```

To read the file, the “fopen” function was used as it was easy to implement as it produced what I wanted it to. To parse the program I created variable strings of char array, which I would use later to check if this sub-string is inside the input line (as the input gets read one line at a time, done it using a while loop until the next line is end of file).

```
                char *bc = "busCapacity";
                char *bt = "boardingTime";
                char *rt = "requestRate";
                char *pi = "pickupInterval";
                char *md = "maxDelay";
                char *nb = "noBuses";
                char *ns = "noStops";
                char *m = "map";
                char *st = "stopTime";
```

If the first character was ‘#’ it was known that it was a comment line, therefore the line was ignored to move down to the next line. The sub-string (strstr) function was used to check if the sub-string of the created variable are present in the line (named ‘c’), and then I would scan the file to file the variable name, and the next element after it I would store in the global variable, as I know what type it would be.

```
 if (strstr(c, bc))
{
       sscanf(c, "busCapacity %d", &input.busCapacity);

}
```

‘sscanf’ function will read the line ‘c’ and it will look for the pattern as next argument, and the value at ‘%d’ place, it will store in the variable which is stated as the third and final argument. In this case it is the global variable. Used the same structure for all the variables, as I found it easy to implement, and it was quite efficient and quick.
The only place where I have used something different is in the ‘map’ variable, where I had to scan each bit separately and separate them from the line and store them in the array which then was put in the map.

After implementing the parser, I have implemented the shortest path algorithm, which for this simulator I used Dijkstra's Algorithm, as it was easy to understand, and it made sense to me when I was implementing it. Means that it assigns every variable as a nonmember (0), and as you want to find distance, it goes by every stop and when the stop gets visited, the stop is changed to member (1) which means that it won’t be visited again. This prevents infinite loops and it produces the quickest path from one stop to another. Nonmember, member and infinity variables I defined at the beginning of the program as they are the global, constant variables, and they won’t change anywhere in the program. But at the same time allow me to use it anywhere in the program. If the program went into the ‘-1’ path (which means there doesn’t exist a path between two stops), then I would assign an infinity value to it, so this path would never be chosen.

```
#define INFINITY 2000;
#define NONMEMBER 0;
#define MEMBER 1;
```

There are other algorithms which could have been implemented, but for me the Dijkstra's Algorithm made most sense to implement.

The requests that were meant to be generated, were suppose to be created using exponentially distributed method, and as it does make sense to me (multiplying the mean of rate multiplied by the log of the random variable between 0 and 1, $-\lambda \cdot \log(\text{rand}(0.0, 1.0))$, where $-\lambda$ is the mean request rate). It didn’t make sense for me on how to implement it, so instead to get the random time for requests, I used a rand() function and I created a random number in range 0 – 60, and stored it in an array where then I repeated a process ‘request rate’ times and ordered in increasing order. They would be the minutes at which the requests would happened, and every hour I would create a set of values, randomly generated for request times, stops and pickup times.

```
void requests(int *req)
{
        int temp;
        int r = (input.requestRate + 1);
        for (int i = 0; i < r; i++)
        {
                req[i] = rand() % 60;
        }
        for (int i = 1; i < r; i++)
        {
                for (int j = 0;j<r - i;j++)
                {
                        if (req[j] >req[j + 1])
                        {
                                temp = req[j];
                                req[j] = req[j + 1];
                                req[j + 1] = temp;
                        }
                }
        }
}
```

In the program when calculating the requests and times, I would use variable ‘i’ as counter and indicate when the request has been done. Knowing the variable ‘i’ is the request number, i created a random stops using the same function, rand(). I took the input of ‘pickupInterval’, to be the delay between the request and the wanted pick up time and used it as a constant in the program.

Knowing all of those informations, I created the whole output, stage process in one method, where I checked and compared the times to know which output to print out, so if the time for arrival, departure, request, pickup, or boarding passenger message was needed. I would use global variables of ‘time’ to print times out.
This implementation is quite inefficient, as to do this, I have created many, many arrays to store all of the needed information.

```
int *lastStop = (int *)malloc(input.noBuses*sizeof(int*));
int *arriveStop = (int *)malloc(input.noBuses*sizeof(int*));
int *freeTime = (int *)malloc(input.noBuses*sizeof(int*));
int *freeHour = (int *)malloc(input.noBuses*sizeof(int*));
int *timeSec = (int *)malloc(input.noBuses*sizeof(int*));
int *busleaveTime = (int *)malloc(input.noBuses*sizeof(int*));
int *busleaveHour = (int *)malloc(input.noBuses*sizeof(int*));
int *busleaveDay = (int *)malloc(input.noBuses*sizeof(int*));
int *busarriveTime = (int *)malloc(input.noBuses*sizeof(int*));
int *busarriveHour = (int *)malloc(input.noBuses*sizeof(int*));
int *busarriveDay = (int *)malloc(input.noBuses*sizeof(int*));

int *done = (int *)malloc((input.requestRate + 1)*sizeof(int*));
int *firstBusLeave = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *busLeaveH = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *busLeaveD = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *nobus = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *inOut = (int *)malloc(2*(input.requestRate +
1)*input.stopTime*sizeof(int*));
int *firstBusArrive = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *busArriveH = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *busArriveD = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *busStopNo = (int *)malloc(2 * (input.requestRate +
1)*input.stopTime*sizeof(int*));
int *arriveBus = (int *)malloc(2*(input.requestRate +
1)*input.stopTime*sizeof(int*));
int *lastBusStop = (int *)malloc(2*(input.requestRate +
1)*input.stopTime*sizeof(int*));
departure = (int *)malloc((input.requestRate + 1)*sizeof(int*));
```

```
int *departureHour = (int *)malloc((input.requestRate + 1)*sizeof(int*));
```

This is not efficient way of doing things, as it takes much more memory as it should be done, but because of the way that I implemented other things, I would have to change everything to be able to implement it in the more efficient way, as due to time constrains, I was unable to do it, therefore it stayed like this. It does provide a correct output and the information that was needed, and it does provide the simulation of the requests. (But as the implementation for the whole bus was implemented, only the requests and assigning it to buses is accurate).
Also the way I implemented it, the bus works more like a taxi service, taking individual passenger and taking them to their desired bus stop before taking another request, instead of working as a bus and taking multiple people on the way, which would be much more efficient than it is currently, working as a taxi system simulator instead of bus.