ECM2433 Coursework Report

Design decisions

Throughout writing this simulator I have made many design decisions that layout the logic for my program, here I will explain some of the more major decisions I have made. I have of course also taken into account the specification, so not all of the code is explained here as it wasn't a decision made my me. However where I have interpreted the specification I will try and state my decisions.

Structs (Header file)

Queue

The queue struct is just that, a queue. It is a first in first out linked list that adds to the beginning and removes from the end. It has 3 functions, 2 which allow it to pop and push items while dealing with pointer and memory allocation independently as well as a newNode function which is part of the main file and handles the formatting for the data being put into a new queue.

Tuple

The Tuple consists of 8 attributes and is used to return data back from the runOneSimulation function. The 8 attributes are connected to the information from each side of the lights (left/right).

Other Structures (C file)

Loops

The main for loop found in runOneSimulation performs all the actions for the 500 time units long simulation. It uses many if statements to check what should occur on each loop as well as a random number generator to check if a new car should spawn. The main while loop found in runOneSimulations is to clear the end of the queue for traffic that is still waiting after the 500 time units have elapsed. In the event that the side of the queue where the lights are green is empty it will no longer change the lights for efficiency. For both of these loops if the cars can go through the lights they always will.

Maths and Logic used throughout

Some of the most complex logic used in the code is about how we should check what happens on each iteration. First it performs check to see if it should change the light, keeping track when it was last changed and what side it is on using a counter called currentTimeInGreen, and an int that represents a bool in our system called lightLeftOn (1 for true if the left side is green and 0 for false if the right side is green). If the lights don't need to be changed it will then go on to decide what will happen next, the moving and spawning of cars. The spawning of

cars will be based on the rate given at the beginning, this will be a number between 1 and 60, we then generate a number between this and if it is below the rate given we spawn a car, otherwise we do not. I have used the number 60 as we are talking about time here and 60 is a common measurement of time, also it is the lowest common factor of 2,3,4,5,and 6 which makes it greats for rate based calculations, 1/2,1/3 etc (60 is 100% chance). We then also use some more calculations to work out the basic mean averages in runSimulation.

GSL

For the random number generator I have chosen to use GSL instead of rand() to get uniform distribution as well as to be able to have a higher range of test results. It is seeded with time and is set up in our main function. I have declared the r and T for the gsl_rng in the global scope and having them as constants instead of variables as passing them through into my functions when I had more than one parameter seems to create a segmentation fault.

Remaining information and variables

In the runOneSimulation function there are variables named trafficInLeft and trafficInRight, these are to control the length of the queue and to ensure that we are never trying to pop something that doesn't exist.

User input for parameters

The specification says that you should get the input from the command line at run time, I have taken this to mean you take them when the code is run, therefore we use the argc and argv variables to take them from the command line, this could be changed however I think it offers an elegant solution as the user can run lots of tests very quickly with one command and not have to be putting in a stream of inputs one by one.

Error handling

Throughout most of the code we don't really require heavy error processing, as the code should be inputting its own data and the logic will take care of the rest. However at the beginning we have some user input when we run the code and therefore we will need to have some validation. For this we simply check that there is the right amount of inputs when the code was run on the command line, if there isn't we output to stderr (errorFile.txt) and then end the program there. If the inputs are of the wrong kind atoi (which converts from the char* to a int) takes care of this and simply gives us 0. Also although we only use 0-60 as traffic arrival rates we can still run with other numbers you will just get the same results as 0 or 60.

Memory handling

The queue uses malloc to deal with its memory delegation. To deal with possible leakage I have freed the memory when popping the item, and given if the code is running correctly the queues should be empty by the end, this should prevent any issues. When running on the servers I had no memory leaks.

Compiling and running

The main instructions for this will be in the read me. I have a compileSim.sh as stated in the specification which will compile and link all of the files together. You can then run the code and if you input the correct data you can get your simulations, otherwise you will get an error as described above.

Files

I have opted for a more simple system of having one file containing all the code for the process and one for the prototypes of the structs and all of the inputs. I felt that given the runSimulations function is so dependant on the runOneSimulation function it would improve readability to have it this way.

Experiments

Simulation Experiments

Left Traffic arrival rate	Left light period	Right traffic arrival rate	Right light period	Left no. of vehicles	Right no. of vehicles	Left average wait time	Right average wait time	Left max wait time	Right max wait time	Left clearance time	Right clearance time
30	3	30	3	189	186	5	5	32	31	15	15
12	3	30	3	75	187	0	5	5	30	0	6
30	1	30	3	166	167	48	0	247	4	83	0
12	1	30	3	66	168	1	0	26	4	1	0
30	30	2	5	194	196	49	0	216	5	84	0
50	30	2	5	325	193	154	0	332	5	215	0
60	30	3	3	375	187	134	5	263	32	202	22
2	60	1	10	13	424	0	20	29	46	11	40
15	15	2	2	83	81	0	0	6	6	0	0
50	50	2	2	276	276	103	104	432	433	428	428

Figure 1 - table of experiment results (full results and this table in excel can be found in the submission folder

I have run a variety of different experiments on my simulator, the results of which can be found in figure 1. The first of these I think is a pretty standard base line test where we have the lights on for 3 time periods and the likelihood of a car arriving is 50% for each time period. This had lead to a pretty balanced result where we have very similar results on each side and overall the traffic wasn't too bad. The next demonstrates what would occur if there was lighter traffic on one side, but the lights were still equally timed. Here we can see that the traffic level stayed the same on the right side but got a lot better on the left side where there was a lot less traffic. We then used the baseline arrival rates (30,30) to test what would happen when you change the traffic light period, here we can see the traffic gets a lot worse as you shorten the time the lights are green, this is pretty expected. After we have seen what happens when you reduce traffic or light period we can combine these tests to see that when you reduce the traffic and lights in parallel to each other we get again pretty good results.

After these very basic tests we are going to perform some more strenuous simulations. From these we can see that increasing traffic on one side and decreasing the time the lights are green makes the traffic a lot worse. Also not increasing the time the lights are green and making the traffic very heavy makes the traffic very bad on one side and stays the same on the other. If the traffic is very unbalanced we can handle this quite well using an increased light period. And finally having light traffic is going to be better than having heavy traffic, which is pretty reasonable but was important to show how much of an effect this would have. These experiments have also allowed me to test my simulation using correct and boundary inputs.

A lot of the experiments I have run would compare well to the real world. Often a set of traffic lights may have varying traffic at each side, so times of the day may be busier for one side or the other, like in rush hour traffic. You could also have lighter traffic at some times of the day than the average, like during the early hours of the morning.

Overall the results of our experiments have told us that if you increase the traffic without increasing the light period for that side the traffic gets a lot worse. If you increase the time period accordingly the traffic only gets a little worse for that side. Having heavy traffic on both sides and increasing one side for the lights means that we get decent results on one side and long wait times on the other. Having less traffic on one side and more on the other can produce good results if the lights timing is reflective of the traffic, and light traffic on either side will often produce better results. Also how often you change the lights will change how many cars spawn as it will be either the lights change or the cars spawn.

Output

```
[xx999@emps-ugcs1 coursework]$ ./TrafficLights 30 30 3 3
Parameter values :
   from left :
      traffic arrival rate: 030
      traffic light period : 003
   from right :
      traffic arrival rate: 030
      traffic light period : 003
Results (averaged over 100 runs) :
   from left :
      number of vehicles : 189
      average waiting time : 005
      maximum waiting time : 032
      clearance time : 15
   from right :
      number of vehicles : 186
      average waiting time : 005
      maximum waiting time : 031
      clearance time : 15
```

Figure 2 - raw output data from a test