ECM2433 C Programming Simple Traffic Simulator Report

Design decision of the queues

Traffic from both directions can be modelled as queues on the left-hand side and right-hand side as typical road traffic activities on planet Earth follows the pattern of the ‘first-in-first-out’ principle. Each vehicle takes up one space in the queue, so vehicles like a motorbike and a car can not be occupying the same space at the same time as what would happen in real life. Therefore, each item in the queue represents a vehicle.

I implemented the queue in the form of a linked list rather than arrays because an array requires a fixed size continuous memory, which can be problematic to allocate more memory when more cars are added to the queue, let alone the operational cost involved to increase the size of an array as it grows during the simulation. In contrast, each item/node in a linked list points to the memory location of the next item, which means it can use memory space efficiently by dynamically allocating memory from any available memory location as the size of the list grows during the simulation, without the need of continuous memory spaces.

For the purpose of this simple road work simulator, each vehicle in the linked list is represented with a single character ‘c’ in each node when added to a queue. The reason is that the data type ‘char’ has the smallest memory space requirement that sufficiently satisfies the implementation of other aspects of this simulator. It is a good practice in C programming to assign sufficient memory space to any variables; Each node in the list also has an integer type variable that holds the time when it is added to the queue, as it is needed to calculate the individual waiting time for each vehicle as it leaves the queue. The ‘int’ datatype is chosen for holding the time variable for each vehicle rather than any other integer data type is because its integer range is neither too small nor westfully too large in the operational context of this simulator. For example, ‘unsigned short int’ holds integer up to 65535, whereas ‘int’ datatype holds up to 2147483647. ‘int’ datatype allows sufficient memory redundancy in case of some edge case operation. Whereas the integer range of ‘long int’ is far too big to hold an unlikely time unit within any reasonable edge cases.

The memory allocation of each node is checked afterwards to make sure memory is successfully allocated. Otherwise, the program terminates to ensure the simulator works as intended.

Choice of randomness

Rather than using the default random library in C, I decided to use the uniform random distribution from the GNU Scientific Library (GSL) to randomly add vehicles to the traffic/queues. This library can provide better randomness on its seeding. Hence, this helps to make the simulation closer to reality.

Design decisions on implementing the runOneSimulation function

The ‘runOneSimulation’ function accepts the four inputs required for the simulation, a pointer to a struct and a pointer to the random number generator from its caller in the program's main function.

The datatype for the traffic rate on both sides is a ‘float’ between 0-100.00. ‘0’ means zero per cent chance of incoming traffic, whereas ‘100.00’ means a vehicle is definitely added to the queue when there is no traffic light switch in every iteration. The decision of using a floating-point number for this input means users can enter any decimal number according to their needs for different simulation demands.

The datatype for the traffic light period can only be an integer because the time unit of this simulator is one iteration of a loop. Therefore, the integer is the only suitable numeral datatype.

I used ‘#define’ to create a Boolean type to aid in implementing this function, so the code is more readable and maintainable for any programmer.

The traffic lights in this simulator are implemented with a ‘char’ datatype variable that holds either ‘0’ to represent green light or ‘1’ to represent red light. Traffic can only show one signal at each time unit, so the ‘char’ data type is a sufficient choice; The initial default value of the left traffic light is set to ‘0’ and ‘1’ for the right traffic. This setup should not affect the quilty of the simulation as their value will eventually change according to the traffic rules during runtime. A ‘boolean’ variable is used to inform the simulator whether an iteration is a round of traffic lights switching or a round of traffic activities. This variable is reset back to false every iteration until the function find out whether this iteration is a round of traffic lights switching or a round of traffic activities after calculating the remainder of the number of iteration and traffic light period. If the remainder is zero, there is switching for this round. Otherwise, there are only traffic activities for this round.

The traffic light period rate does not need to be the same in both queues. A collision condition is created to handle the scenario when both lights switch to the same colour. Only the queue with the longest average gets a green light when their lights are the same. This design allows a fairer amount of traffic activities to reduce the overall average waiting time on both sides.

Vehicles are randomly added to the queues when the iteration is not a light switching round until 501 iterations are reached. When adding a vehicle to the queue, a node with a character ‘c’, the number of current iterations, and the pointer to the last item on the linked list are assigned to its attribute. At the same time, a front node is removed from the queue when its traffic light is green. Upon removal from the queue, the waiting time of each node is calculated by the difference between the current iteration number and the time of creation. The waiting time of each vehicle is added to a total waiting time variable outside the scope of the while, and the average waiting time of a queue later is calculated when the loop is finished. The total number of vehicles and the current maximum waiting time of both queues are updated upon removing every single vehicle from the queue.

A counter with the ‘int’ data type is used for holding the iteration number for the while loop as it gets incremented after each iteration.

Since each simulation performs at least 500 iterations, a while loop condition is chosen rather than a for loop because it requires a definite amount of loop, which is not suitable for the use case of this function. The while loop condition makes sure it only stops when both queues are empty after 500 iterations, and the timers on clearing the queues have been used.

The timers start at iteration number 501, as the specification mentioned ‘after 500 iterations’. Two conditions are added to trigger timers to avoid re-triggering the same timer after its queue is empty for the rest simulation. The timer of a queue stops when its queue is empty. The clearance time of a queue is calculated with the difference between the iteration number at the stop and the iteration number at the start.

The simulation is complete at the end of the while loop, so the queues are free from the memory space.

The result data of a single simulation is stored in a structure with a fixed amount of variables created from the caller’s main function with a pointer passed into the function via the parameters. The same can be done by passing a pointer to an array, allowing resizable data storage. However, this is not useful for the context and design of this function as it only needs to transfer eight variables constantly back to its caller.

The struct stores all the results in ‘float’ datatype because they are either result of division or data that is yet to be calculated with division from the main function.

Design decisions on implementing the runSimulations program

According to the specification, the main function in the ‘runSimulations’ program takes four arguments from the command-line input. Error message conditions are created if an invalid input and amount of arguments are supplied. A usage suggestion message is also displayed after the error message with an example of the usage. This condition only suggests an input range for each input but does not restrict any out of bound input as it allows users to perform any kind of experiments after they are informed.

A GSL random generator with a unique seed is generated before running multiple simulations. A pointer is created that allows every simulation to obtain a random number back from the generator in the main function. This design decision means all single simulations continuously use the same seed but with different parts of it.

The variables used as counters are also initialised before running the simulations, later used to calculate the average over 100 runs.

A for loop is used to run all the required amounts of a single simulation as the number of loops is specified. In the for loop, a structure with a type defined as ‘RESULT’ is instantiated at the beginning according to the memory allocated. Each memory allocation is checked after using ‘malloc’ to ensure there is enough memory to store the data for the results of each simulation. The program exits if there is insufficient memory space to run.

The results after each run are added to the counter in the main function, and then the memory allocated for the ‘RESULT’ struct is freed after each loop. Freeing memory space when a data structure is no longer needed is a good practice in C programming.

Design decisions of the program files structure

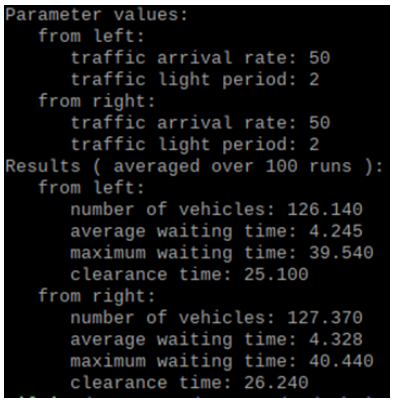
The code for the simulator program consists of three parts, the queue, the single simulation and the run simulations program. Each part of the code is further divided into header files and C files.

The header file for the queue contains all the ‘#includes’, structure definition of the queue, and function prototypes needed for the function definitions in the C file for manipulating the queue, like functions that create, add and remove nodes to and from the queue. Most of the functions are set to the public scope so they can be accessed externally.

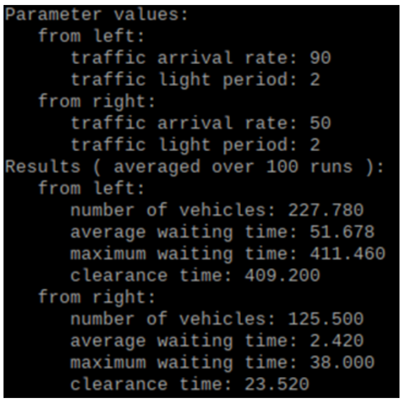
The header files for the single simulation contain all the ‘#includes’, structure definition of storing the results, and function prototypes needed for the function definitions in the C file for manipulating the simulations, like functions creating result storage structure, running a single simulation, switching the lights, and outputting the results in a ‘stdout’ stream according to the suggested format from the specification. Most of the functions are set to the extern scope so that the simulator program can access them.

Separated files mean reusability, modularity, and maintainability for the codes so developers can easily make changes, update, or even reuse part of the code on another project. This is generally a good practice in programming in a general way.

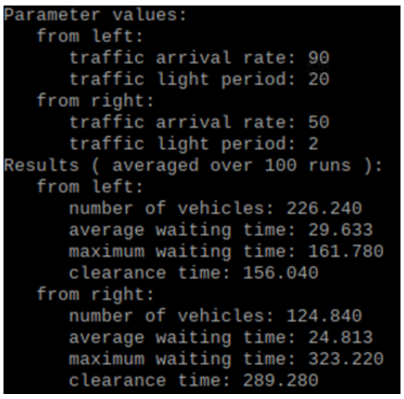
Experiments and Results:



```
Parameter values:
   from left:
      traffic arrival rate: 50
      traffic light period: 2
   from right:
      traffic arrival rate: 50
      traffic light period: 2
Results ( averaged over 100 runs ):
   from left:
      number of vehicles: 126.140
      average waiting time: 4.245
      maximum waiting time: 39.540
      clearance time: 25.100
   from right:
      number of vehicles: 127.370
      average waiting time: 4.328
      maximum waiting time: 40.440
      clearance time: 26.240
```



```
Parameter values:
   from left:
      traffic arrival rate: 90
      traffic light period: 2
   from right:
      traffic arrival rate: 50
      traffic light period: 2
Results ( averaged over 100 runs ):
   from left:
      number of vehicles: 227.780
      average waiting time: 51.678
      maximum waiting time: 411.460
      clearance time: 409.200
   from right:
      number of vehicles: 125.500
      average waiting time: 2.420
      maximum waiting time: 38.000
      clearance time: 23.520
```



```
Parameter values:
   from left:
      traffic arrival rate: 90
      traffic light period: 20
   from right:
      traffic arrival rate: 50
      traffic light period: 2
Results ( averaged over 100 runs ):
   from left:
      number of vehicles: 226.240
      average waiting time: 29.633
      maximum waiting time: 161.780
      clearance time: 156.040
   from right:
      number of vehicles: 124.840
      average waiting time: 24.813
      maximum waiting time: 323.220
      clearance time: 289.280
```

Picture 1: Experiment 1   Picture 2: Experiment 2   Picture 3: Experiment 3

Experiment 1's setup: Same arrival rate on both sides with the same traffic period

The overall results from both queues are very much similar. This suggests that both queues operated with similar traffic and light switching activity in the simulation due to the consistent uniform randomness provided by the GSL random number generator setup.

Experiment 2's setup: Different arrival rates on both sides with the same traffic period

The overall increase in the left queue's result shows the effect of an increased traffic rate has on each category of the results. This drastically increases the total number of vehicles, the average waiting time, maximum waiting time, and clearance time of the left queue. Whereas the results on the right queue are relatively similar to the right queue in the first experiment.

Experiment 3's setup: Different arrival rates on both sides with different traffic periods

The results from both queues are much different from the results in the prior experiment. The left queue performed better than the right queue with a greater overall input, although the right queue has the same setting but relatively smaller input. The left queue's traffic rate is the same as in experiment 2, which explains the similar total number of vehicles in the queue. An increased rate of arrival vehicles might contribute to its higher average waiting time. Its lower maximum waiting and clearance time could be affected by a long traffic light period that allows more cars to leave the queue when green.

Other experiments on known runtime errors

Suppose the user decides to ignore the usage suggestion and enter any non-numerical value—for example, characters and symbols. The program would throw a 'Floating point exception' before it terminates. The cause of this error is that the program tries to convert a character input to a numerical value, which will fall because the conversion function does not take any character input in its arguments. I did not go further to create an input validation for this matter because input validation is not significantly a major for this simulator program in the specification. There are already a sufficient number of warning messages in the existing input validation setup.