Discrete Event Simulation

Note: Due to my partner dropping the course, I worked on the application portion with no partner to work on the simulation engine with me. Therefore, apart from a few modifications that I made, I used the sample simulation engine given to me.

- DES is a very interesting way of simulating real-life scenarios and environments at discrete iterations. A key element of this system is that all events are being performed in a defined order. The simulation engine deals with "holding onto" events scheduled by the simulation application. When the current time is equal to the timestamp of the scheduled event, then a callback function is called using the EventHandler to allow the simulation application to then initiate the event. The .h files provide an API for which the engine and application can follow the same function protocols. The engine should be able to identify these without being able to see the inner workings of the application. The application can the imitate any real-life situation using different types of events and states.

- There were no major changes to the API from the sample simulation, apart from adding in the two rand() method signatures

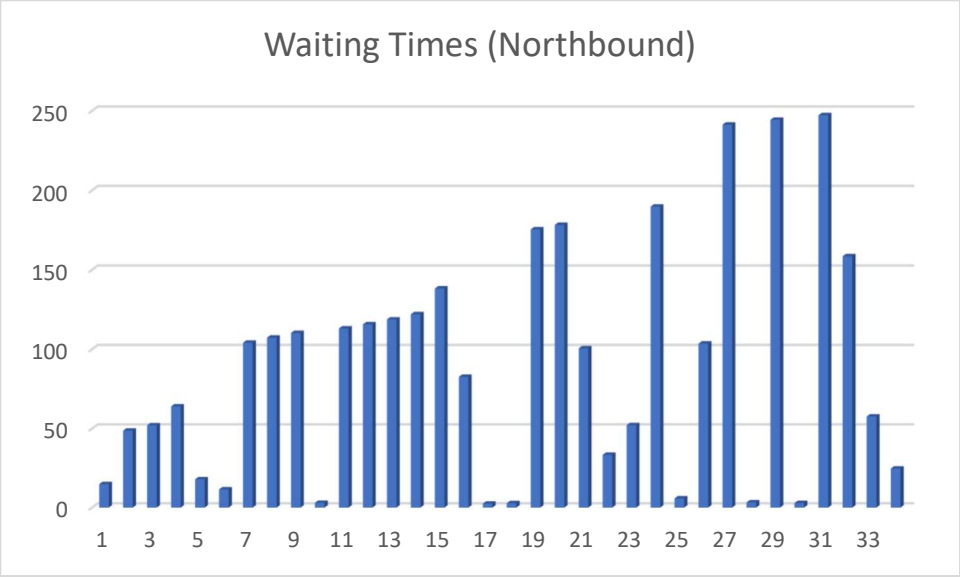
- The simulation application consists of 3 state variables, namely Empty, South, and North. These state variables were defined as integers to make it easy to compare state (integers seem slightly easier to compare than strings in C). There were 6 total event variables. In particular, there were three distinct ones that applied for both northbound and southbound (the nice thing about this simulation is that the application is symmetrical, meaning the rules apply identically for North and South). The 3 distinct events are arrival (getting to the bridge), start (getting on the bridge), and finish (getting off the bridge). The arrival event contained multiple cases and was the longest function to implement. I had to account for all the cases where either north or south had a line of cars waiting combined with which state the bridge was in.

    - Sample Pseudo Code:

```
void northArrival (struct EventData *e) {
        assign a block of memory for event
        set event type to desired type (as in method signature)
         if statement (cases mentioned above):
                assign current time values to attributes of the car (ie startTime,
                waitTime) by using rand functions to add random amounts of time
                Then, re-set the event-type to the one following this event and schedule
                it
```

  The remaining functions start and finish are shorter because they are not as complex. They serve mainly to increment the values of group size and track time.

- I tested my code by printing out the state, timestamps, and group size at each iteration. I also separated the data from the northbound and southbound traffic to make sure that my output was symmetrical. Below is a graph showing the traffic waiting time for one run.



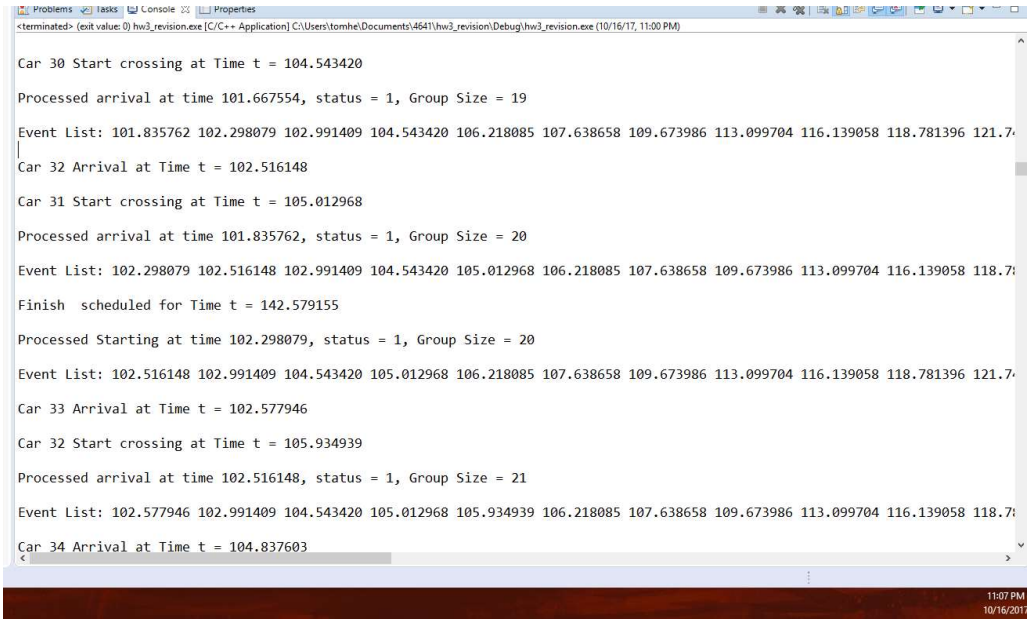
Here are some sample run-times that show that the average waiting time gradually increases as simulation time is extended.

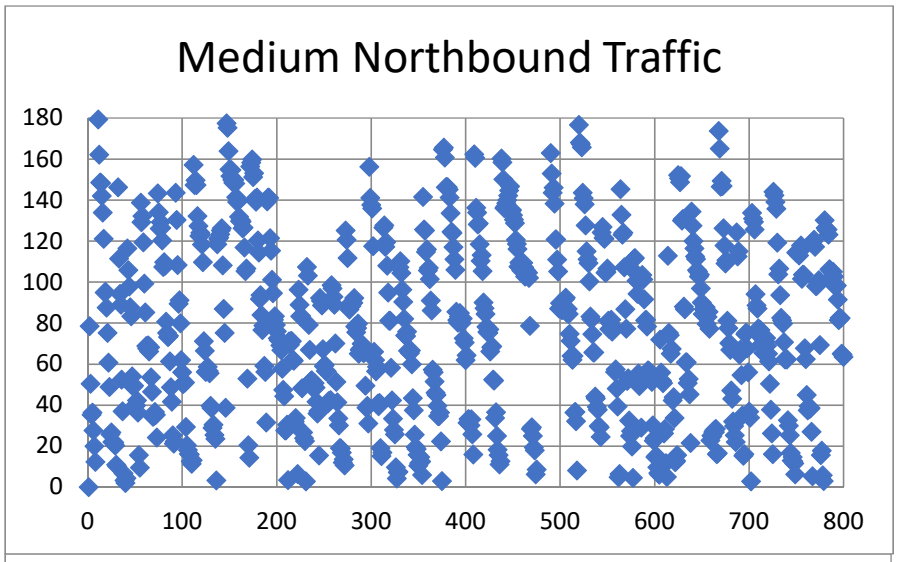
RunSim(300.0) had a minimum waiting time of (2.59 seconds), a maximum waiting time of (299.66 seconds), and an average waiting time of (68.18 seconds).

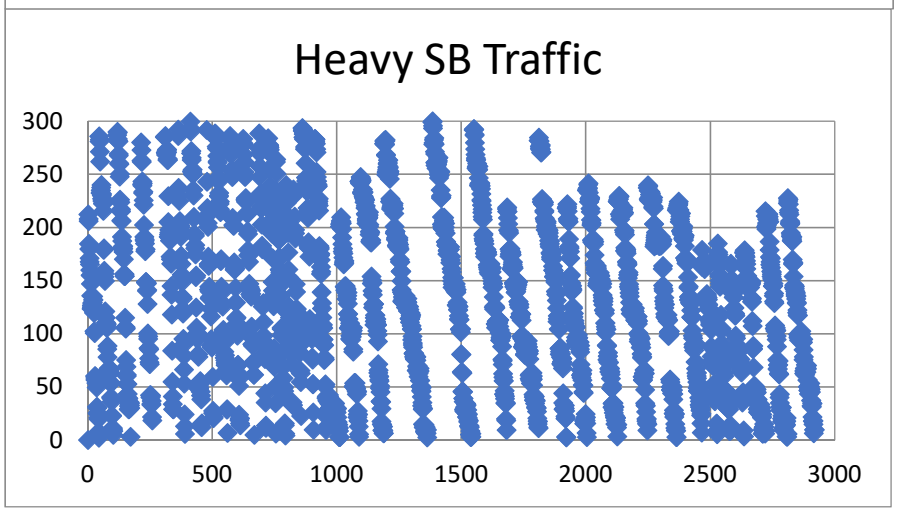
RunSim(200.0) had a minimum waiting time of (2.59 seconds), a maximum waiting time of (198.19 seconds), and an average waiting time of (64.30 seconds).

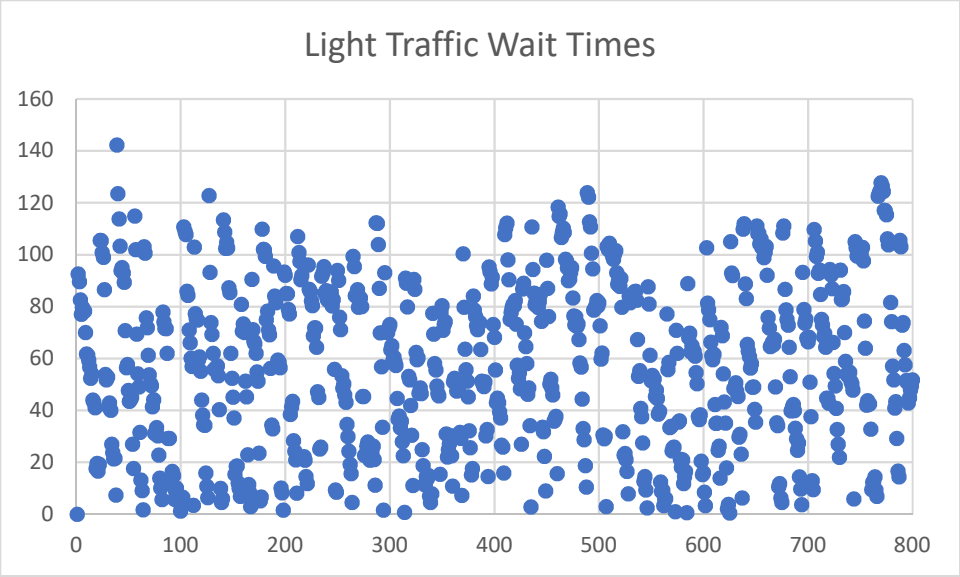
RunSim(400.0) had a minimum waiting time of (2.55 seconds), a maximum waiting time of (322.98 seconds), and an average waiting time of (91.15 seconds).

Below is an example of the raw data output I would get for each simulation run.




Medium Northbound Traffic
180
160
140
120
100
80
60
40
20
0
0
100
200
300
400
500
600
700
800


Heavy SB Traffic
300
250
200
150
100
50
0
0
500
1000
1500
2000
2500
3000



- I cannot cover bullets 5-7 because I have no partner. I was only able to run the simulation with the sample engine code (which I slightly modified).

- I noticed that the slope between end time and wait time in queue is mostly flat. Although, there were spikes and pits (in wait time) at certain points in the simulation. This is due to using the exponential distribution to schedule events. Sometimes, events will happen to be more spaced out. At other points in time in the simulation, the exponential function caused the application to schedule multiple event right after each other. Like in the first graph shown, traffic is somewhat constant but comes in waves (high to medium or medium to low) depending on when a car arrives at the bridge. This takes the shape of a cosine function shifted vertically upwards.