CX 4010 / CSE 6010 Assignment 3: Discrete Event Simulation

Due Dates:

- Due: 11 AM, Friday, October 19, 2018
- Revision (optional) Due: 11:59 PM, Monday October 22, 2018
- No late submissions will be accepted

1. Background

Queueing networks are widely used to model systems where customers must utilize some service, and wait if the service is being utilized by other customer(s). For example, in a fast food restaurant, the restaurant employee standing behind the counter is modeled by a server who must handle customers waiting in a queue to place their order and receive food. Such models are widely used to analyze health care systems, air and road transportation networks, businesses such as restaurants or department stores, manufacturing systems, and computers and communication networks, to mention a few. All of these systems involve *customers* (e.g., people, aircraft, vehicles, data packets, computer jobs, parts) that travel from one *station* to another to receive service at each station. Each station includes a server and a queue to hold waiting customers. Here, we assume each station can only process one customer at a time. Customers wanting to use a server that is busy handling another customer must wait in the queue until the server is available to process another customer.

This assignment involves developing a simulation model for a system of queues, and using it to analyze a hypothetical cafeteria. You will work in teams of two students each to complete this assignment with each person in the team required to implement a significant portion of the simulation. An important part of the assignment concerns dividing up the software development into two parts and defining simple, clear interfaces between them so each person can develop their own software without having to become overly familiar with implementation details of other parts of the software.

2. The Queueing Network Simulator: CPS-Sim

The simulator dubbed *CPS-Sim* is a general-purpose queueing network simulator that can be used by others with no ability to program. It will include four different types of components, enumerated below. Each component has one or more parameters. Each component may have some number of input and out ports (possibly zero) through which customers arrive and depart. The components are (see Figure 1):

- *Generator (G): creates new customers and injects them into the system*. The generator component (also called a source in the literature) has a single output port. The component has a single parameter indicating the average time between the arrival (generation) of successive customers, i.e., the interarrival time. Assume the interarrival time is drawn from a random number generator following an exponential distribution.
- *Exit (E): a sink to remove customers from the system*. Each customer arriving at an exit component is immediately removed from the system. This component has no parameters.
- *Queueing Station (Q): a server and a queue holding waiting customers*. The station's server processes customers, one at a time. If the server is idle when a customer arrives, that customer immediately receives service. If the server is busy, the customer is placed into a FIFO queue and waits for the server. Once the server completes processing a customer, it provides service to the first customer in the FIFO queue, if there is one. The queueing station has one input through which customers arrive, and one output port through which

customers depart after they have received service. This component has a single parameter that indicates the average time required to serve a customer; a customer remains in a queueing station for this service time plus the amount of time (if any) that it must wait in the FIFO queue. Here, service times are drawn from a random variable following an exponential distribution.

- *Fork(F): a router that sends customers to another component*. A fork component has one input on which customers arrive, and *K* outputs. Each arriving customer is routed to exactly one of the *K* outputs. A customer is routed to output port *i* with probability P_{ii} ($\sum_{i=1 \text{ to } K} P_i =$ 1). Each fork component has *K+1* parameters, namely the number of output ports *K* and the *K* probability values.

Assume customers require a negligible (i.e., zero) amount of time to travel between components, or to travel through a fork component.

A sample queueing network is shown in Figure 1. The generator creates a new customer with average interarrival time of 15 units of time. From the generator the customers flow to service station Q_1 with average service time of 20. The customer then is routed to Q_2 with probability 0.3, or to Q_3 with probability 0.2 before traveling back to Q_1. Alternatively, the customer may be routed directly back to Q_1 with probability 0.1 or may leave the system with probability 0.4.

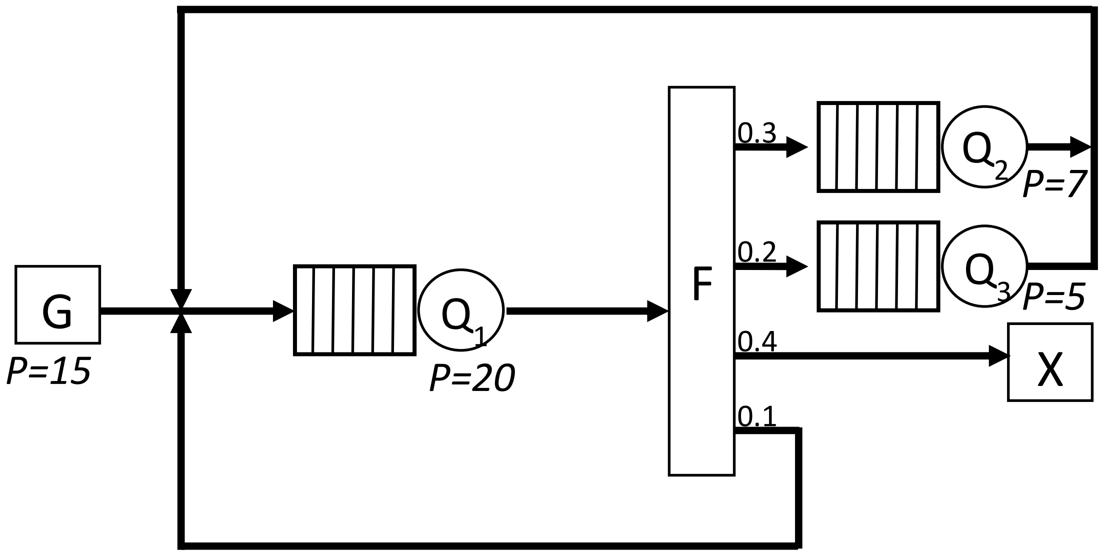

Figure 1. Sample queueing network.

3. Configuration File

The particular queueing network that your software will simulate is specified to a *configuration file* that specifies the components, their parameters, and outgoing connections to other components. The simulation program will first read this configuration file, instantiate the queueing network and then run a simulation of the network.

The format of the configuration file is as follows:

1. The first line of the file should contain a single integer indicating the total number of components in the queueing network. If there are N components, each is assigned an integer ID with ID values 0, 1, … N-1.
2. The remainder of the file contains N additional lines, with each line describing a single component and its connections to other components. Each line contains a number of fields, separated by one or more blank spaces. The first two fields of each line indicate the component ID (0, 1, … N-1) and a single character indicating the type of component (`G` for generator, `E` for exit, `Q` for a queueing station, and `F` for fork). The remaining fields in the line indicate component parameters and the ID(s) of the component(s) to which this component may send customers. These depend on the type of component. Average interarrival times, service times, and probabilities for fork components should be specified as floating point values while component IDs are integers. The specific format of each line is as follows (note that the brackets < > do not appear in the file:
    a. *Generator*: `<ID G P D>` `ID` is the component ID and `G` indicates this is a generator component. `P` indicates the average interarrival time. `D` is the ID of the component to which generated customers are sent.
    b. *Exit*: `<ID E>` `ID` is the component ID and `E` indicates this is an exit component.
    c. *Station*: `<ID Q P D>` `ID` is the component ID and `Q` indicates this is a queueing station component. `P` indicates the average service time. `D` is the ID of the component to which departing customers are sent.
    d. *Fork*: <ID F K P_0 P_1 … P_{K-1} D_0 D_1 … D_{K-1}> `ID` is the component ID, `F` indicates this is a fork component, and `K` indicates the number of output ports on the component which are labeled 0, 1, … K-1. The next K fields are probability values, with P_i indicating the probability a customer is routed to output port i. These K probability values should add up to 1.0. The K parameters that follow are integers indicating the ID of the components connected to the output ports, i.e., D_i gives the ID of the component connect to output port i.

For example, the queueing network shown in Figure 1 is represented with the following configuration file:

```
6
0 G 15.0 1
1 Q 20.0 2
2 F 4 0.3 0.2 0.4 0.1 3 4 5 1
3 Q 7.0 1
4 Q 5.0 1
5 X
```

4. Software

You will build a discrete event simulation (DES) program to simulate the queueing network. The software you develop must include two separate parts (the source code for these must reside in different files) that will be compiled together to create an executable simulation:

1. *Queueing network simulation library*. This software is a library of functions for creating and executing simulations of queueing networks. The library should be structured as a general-purpose queueing network simulation library not specific to a particular network configuration or application.

2. *Configuration program*. The configuration program is responsible for reading the configuration file, checking it for errors, and using primitives defined in the queueing network library to create the queueing network specified in the file. It then uses a primitive provided by the library to execute a simulation of that network. After the execution is complete it produces various output statistics.

You must define an interface to the simulation library that hides as much as possible the internal implementation of the library. You should minimize (ideally, complete avoid) any variables that are shared between the library and the configuration program. The interface to the simulation library must be clearly documented in a `.h` file. At a minimum, the simulation library interface should include the following functions:

1. Functions to dynamically (i.e., during the execution of the program) create new components. Define one function for each component type that includes function parameters corresponding to the parameters of the component.
2. A function to run the simulation of the network that was created.
3. Functions to extract information (e.g., statistics such as station waiting time) as needed.

You may need to define other functions. The details of the above primitives, and any other functions needed in the interface are up to you and your partner.

Further, the implementation of the simulator must adhere to the following rules:

1. You must implement an event-driven, discrete event simulation, *not* a time-stepped simulation.

2. Use a double precision floating point number to represent simulation time.

3. Storage for components and events must be allocated dynamically by calling `malloc()`, and the storage released when you are done using the memory by calling `free()`. Programs that have memory leaks or dangling pointers will be considered erroneous!

4. Similarly, `malloc()` must be used for each customer to create storage that holds information concerning that customer, e.g., statistics such as time in the system. This storage should be allocated when the customer is created and released when the customer exits the system.

5. Each event and information concerning a customer must be implemented using a C `struct`. Each event must include a timestamp value and any parameters you deem necessary to characterize the event.

6. To generate random numbers from an exponential distribution with mean `U`, create a function `double urand(void)` that returns a random number uniformly distributed over the interval [0,1) (note it cannot return the value 1.0), then define `double randexp()` that returns `-U*(log(1.0–urand()))` where `log()` is the C function defined in `<math.h>` to compute a natural logarithm.

Your simulator should take as command line parameters (1) a value indicating the amount of simulation time the simulator should run (2) the name of the configuration file, and (3) the name of the file in which the output from the simulation is stored. For example:

```
% cpssim 1000.0 config outfile
```

will execute the program `cpssim` that create the simulator specified in the file `config`, runs the simulator for 1000 units of simulation time, and writes the resulting statistics into `outfile`. The format of the output file is up to you, but it should be designed for people to read, and readily understand the results of the simulation run.

The configuration program that reads the configuration file and creates the queueing network must be robust, able to detect errors and print useful information concerning the error. It is up to you to determine what types of errors might occur, and write this program to detect them and provide useful information to the user to fix the problem.

5. Sample Simulation Software

A sample discrete event simulation (DES) program is provided. You may reuse any portion of this code in your own software, but be sure to acknowledge the source of code you use through comments in the code as appropriate.

Rather than using a time-stepped approach where simulation time advances from one clock "tick" to the next, discrete event simulations advance time from one *event* to the next, where each event represents something "interesting" occurring in the actual system, i.e., the *physical system*. For example, typical events might be the arrival of a new customer in a simulation of a department store or a vehicle arriving at an intersection in a traffic simulation. Each event contains a timestamp, a simulation time value indicating when that event occurs in the physical system. The timestamp is analogous to the time step number in a time-stepped simulation, however, events occur at irregular points in time, not at regular, periodic time steps. Simulation time advances at irregular intervals as the computation proceeds from one event to the next.

The simulation includes a number of state variables that represent the current state of the system, e.g., a queue of customers waiting to use a server or the state of a traffic signal. The computation consists of a sequence of event computations. Each event computation can (1) modify one or more state variables, and/or (2) schedule one or more new events into the simulated future. For example, when modeling air traffic at an airport, the computation for an event denoting that an aircraft has just touched down on the runway might schedule a new event five minutes into the simulated future to represent the aircraft arriving at the arrival gate and beginning to unload passengers.

The simulation program (implemented by the simulation library) contains two main part:

- *Simulation engine*. This part of the program is independent of the simulation model, i.e., the same simulation engine software could be used to simulate a transportation system or a manufacturing application. It includes a data structure called the *future event list* (FEL) that is a priority queue that contains the set of events that have been scheduled, but have not yet been processed. The simulation engine holds the main *event processing loop* which repeatedly (1) removes the smallest timestamped event from the FEL, and (2) calls a function (defined in the simulation application, discussed next) to simulate that event. The loop continues processing events until some termination condition is met, e.g., simulation time reaches a certain value. The simulation maintains a *clock* variable indicating how far the simulation has advanced in simulation time. It also includes a function called by the simulation application to schedule a new event, i.e., to allocate memory for the event, fill in various parameters, and insert the new event into the FEL. The main event processing loop updates the clock variable in each iteration of the loop to the timestamp of the event it just removed from the FEL.
- *Simulation model*. This part of the program contains code to model the physical system. It includes a set of state variables that represent the current state of the system being modeled. It also includes one or more functions or *event handler* procedures, one for each type of event modeled by the simulation. The event handler procedures collectively model the

operation of the system being simulated. To develop the simulation application, one must define the state variables and the different types of events that are modeled, and implement a function for each different event type.

This simulation model in the example code implements a specific discrete event simulation model, specifically, the operation of a simple gasoline station. Although this application is a queueing network, the code is provided for illustrative purposes to understand how a discrete event simulation works. It does *not* conform to the requirements of this assignment. We recommend you completely rewrite this code. Some of the event types used in this sample simulation will likely be similar to the kinds of events your simulation code must use.

6. Simulation Study

Once your software is running, construct a simulation to analyze the operation of a food court. In the following all time values are in minutes. The food court includes:

1. A queueing station where customers enter and receive a tray to carry their food; assume the mean service time for this station is 0.2 minutes.
2. Five queueing stations (A, B, C, D, E) each serving a different type of food. Assume the popularity of these stations (i.e., the probability a customer will select that station) are as follows: A: 0.10; B: 0.15; C: 0.20; D: 0.25; E: 0.30. The average service time of these stations are: A: 3.0; B: 2.5; C: 3.3; D: 2.8; E: 1.50.
3. Two beverage stations where beverages are obtained, after obtaining food. Assume all customers purchase one beverage and each customer is equally likely to select either station. The average service time of the beverage station is 1.0 minute.
4. One or more check out stations where customers must pay for their food. Assume a customer is equally likely to select any check out station. After departing the check out station, most customers (95%) will exit the system. However, the remaining customers (5%) will find they do not have sufficient funds to pay for their food. In this case, their food order will be left at the check out station, and the customer will go back to make another food order. Specifically, the customer will bypass the station for picking up a tray, randomly select one of the five queueing stations (using the same probabilities as reported earlier) and proceed directly to that station to get a new order of food and then get a new beverage.

The following statistics are of interest:

1. Number of customers entering and exiting the system during the simulation run. Note these values will usually not be the same because there will be some customers still in the system when the simulation ends.
2. The minimum, maximum, and average amount of time a customer remains in the system among those customers who exited during the simulation run.
3. The minimum, maximum, and average amount of the *total* time each customer must wait in queues in traveling through the system.
4. The average waiting time experienced by customers at each station/queue. This information can be used to identify bottlenecks in the system.

You should complete several simulation runs varying the rate (the inverse of the interarrival time) of customers entering the system. You should plot curves of (2) and (3) above for different numbers of check out stations in order to determine how many stations should be utilized. Set the length of the simulation run to 4 hours (240 minutes). In an actual simulation study, you would perform each simulation experiment multiple times with different random number streams (initial seeds), however, here, you need only complete one run for each experiment.

7. Teams and Timeline

You will work in teams of two students each to complete this assignment. We will assign students to teams. For each team one student should implement the simulation library, and the other the configuration program. Beyond that, it is up to your team to decide who does what.

You have three weeks to complete this assignment. We recommend that you spend the first week becoming familiar with discrete event simulations (a document is provided) and the sample simulation software, figure out how to divide up the work between team members and who will do what, and jointly develop the simulation library API. We recommend you develop the simulation software in the second week and get it to work on some test configuration files that you should define. Use the third week for experimentation and writing the report.

Don't wait until the last week to complete the entire assignment!

8. Project Report

The project report should be a single document developed by your team. It should be self-contained, and understandable by someone not familiar with the assignment. It should include a short introduction section describing the simulation in high level terms and the goals of the study. The API to the simulation library should be described, as well as description of the implementation of the library and configuration program. It should include a description of the testing procedure and evidence the code works correctly. Describe the results of the experiments that were performed, your analysis of the results (do they make sense?), and explanations of what you observed. Be sure to discuss any anomalous results and explain what happened.

9. Student Taking CX 4010

If you are a CX 4010 student, you must implement the simulation software, collect the results of the simulation runs, and produce a report documenting your results. Examine the results you obtain and argue why you believe your simulator is functioning correctly. You must provide evidence your code works correctly. Pay special attention to your results for high arrival rates.

10. Students Taking CSE 6010

If you are a CSE 6010 student you must complete the steps described above for students taking CX 4010. In addition, validate that your software is producing reliable results. To do this consider the operation of a single queue with a generator to create customers, and exit component to remove customers once they exit queue. This type of queueing network is called an M/M/1 queue and has a known mathematical solution. Compare the mathematical solution with the results produced by your simulation by showing a graph of waiting time as the arrival rate is varied. Write up your results, including references to the relevant literature, and include in your report.

11. Collaboration, Citing, and Honor Code

Finally, we remind you of class policies regarding collaboration, citations, and the honor code.

All students are expected to follow the Georgia Tech Honor Code. You are encouraged to work together and help each other on assignments and preparing for exams, but ***all work you turn in must be entirely your own work***.

We encourage discussion with other students, but all code you turn in must be written entirely by you. We recommend you destroy any notes taken while consulting with other students before writing your own code. For team programming projects each student you must clearly document what code was written by each student; for example, each student's code may reside in separate files. Dissemination of code to other students is not allowed, except in the case of students working together on the same team in team programming projects.

The Internet is a useful resource to solve specific programming problems. You are encouraged to use the Web for information or to answer specific questions, but *copying or utilizing code from the Web violates the Honor Code*. If you use the Web or other sources for any information, you must cite these sources in your assignment submission. If you are unsure what is permissible, consult with the instructor or a TA.