CSCI 204 – Data Structures & Algorithms

Project 3 – Discrete Event Driven Simulation

Phase 1 due: 11:55 pm Monday, April 10th
Phase 2 due: 11:55 pm Monday, April 24th

1 Objective

The main objective of this assignment is for you to practice and master priority queues and to become familiar with discrete event driven simulations.

2 Introduction

You are just hired by a big retail store to study how cashiers should be configured given the statistics from last holiday season so that the store can maximize the profit, yet the customers don't have to wait in too long a line for their services. For online stores, the situation is similar in the sense that you need to decide how much computing power and storage to be allocated for your web servers so that they can serve the customers well, yet the equipment dedicated to the service don't go wasted. To help run what-if scenarios to answer "how many cashiers should there be?" we will build a computer simulation.

Simulation is a widely used technique to model and study the behavior of many physical systems similar to the one described above. These systems may include traffic control systems, computer networks, bank operations, supermarket check-out counters, and many others. In **event-driven simulation**[1], the simulation is driven by the sequence of events that are taking place in the system. That is, the simulation system takes actions only when an event is happening. Specifically, you will build a **discrete event-driven simulation** because the time when events could happen is discrete, not continuous. That is, an event takes place at a discrete time point, e.g., 3, 5, 20, not in a continuous range, e.g., (3, 4].

In event-driven simulation, event timing can be determined on mathematical models, randomness, and in reaction to other events. Every time an event occurs, the program is designed to have a mechanism to handle this type of event. For example, if we take a snapshot of the operations of a bank branch office, we may see at time 10:13 a.m., a customer arrives (Event A), the customer receives the service from a bank

[1] Another type of simulation is **time-step simulation**, driven by time steps. At each clock tick, e.g., minute or second, the current status of the system is examined and events are handled as they occur at that moment of time. In this type of simulation, the program has to step through every tick of time, whether or not anything is happening at that moment. This is not very efficient computation.

teller at 10:17 a.m. (Event B), the customer completes its transaction at time 10:25 a.m. (Event C), and the customer leaves the branch office at 10:32 a.m. (Event D). Figure 1 illustrates the four events that take place in a time sequence.

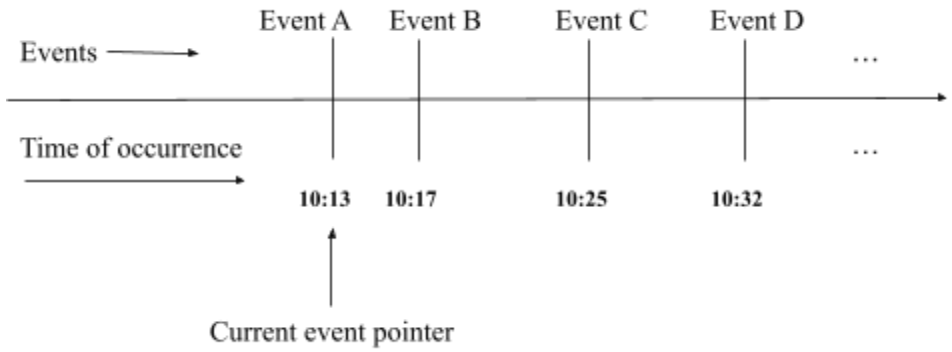

Figure 1: Event driven simulation timeline.

Instead of checking at every moment, e.g., every minute, if something happens in the system, event-driven simulation schedules events to take place at a certain time and puts these events in an *event queue*. The event queue is ordered by the time when the events are happening. *Thus an event queue is a priority queue using the event time as the priority value.* The simulation runs by taking events off the event queue and processing the event accordingly. Processing an event may include generating new events to be added into the event queue for a later time. For example, when the first customer arrives at the bank office, four events caused by this arrival can be inserted into the event queue (Event A, B, C, and D). Assume a new customer arrives at time 10:20, then this arrival event (Event A1) should be inserted into the event queue after Event B, but before Event C.

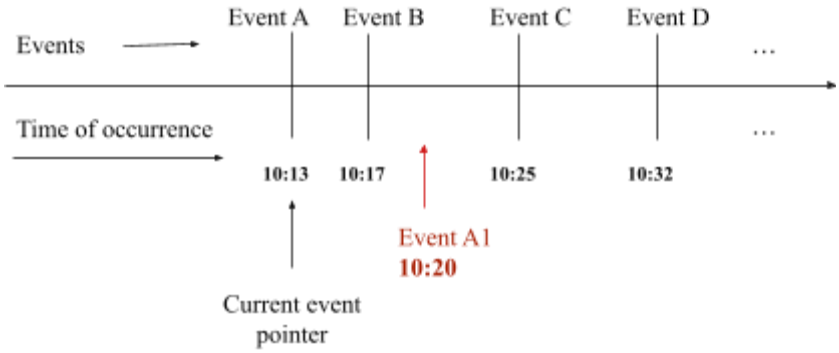

Figure 2: A new customer (A1) arrives at 10:20 a.m.

The general logic of event driven simulation is for the simulation program to process events as they come and schedule future events as appropriate. A general algorithm of a discrete event simulation can be outlined as follows:

```
generate initial events and insert them into the event
queue
while event queue is not empty:
    take an event off the event queue
    set the simtime as the event time
    if simtime < totaltime:
        process the event (possibly generating new events)
```

Figure 3: General program flow of discrete event driven simulation.

3 Simulating Cashiers in a Retail Store

Your task is to write a discrete event driven simulator to simulate the behavior of cashiers at a department store when the customers come to the store in a certain statistical pattern. The project is divided into two phases. In the first phase, the event queue (a priority queue) is implemented using a linked list. In the second phase, the event queue will be using a heap (a more efficient data structure to implement the priority queue).

Simulation Overview

Our simulation has for settings:

- `num_cashiers`: the total number of cashiers, this does not change during the simulation
- `max_inter_arrival_time`: the maximum number of time steps between customer arrivals. We will use this to model customer arrival.
- `max_service_time`: the maximum number of time steps it will take for a cashier to check out a customer. We will use this in our statistical model of service times.
- `total_sim_time`: the total number of steps in our simulation

Our simulated store will have `num_cashiers`. When a customer arrives, if there is an open cashier, the customer will begin getting service. If all cashiers are occupied, the customer will enter a waiting queue (a standard queue). When a cashier finishes serving a customer, they will serve the next customer in the waiting queue, if there is one.

Queues

Two types of queues are needed for the simulation. Both have been initialized as attributes in the `StoreSimulation` class:

`events - Priority Queue:`

Events that happen in the simulation are stored in the event queue. The events on the even list are ordered by the time when the event should take place in the simulation. In other words, an event's priority in the queue is based on its time. (See Figure 2 for an illustration of events happening on the timeline.) The main simulation loop (See Figure 3) always takes the first event off the event queue and processes it as the current event, until either the event queue becomes empty, or the total simulation time is reached.

For Phase 1, you will implement linked list-based priority queue in `linkedpriorityq.py`.

The `PriorityNode` class has three attributes:

- `data`, an Event object (see the next section)
- `priority`, an integer representing the time of the event
- `next`, None or the next PriorityNode object in the list.

☑ ~~***Complete the PriorityQueue class in `Linkedpriorityq.py`.***~~

☑ ~~As you complete the class, add code to main.py's method test_priority_queue() to ensure your priority queue implementation is correct before moving on.~~

`waiting - Standard FIFO Queue:`

All customers who arrive but can't start the service immediately should be put into a waiting queue for the cashiers. All customers share one common waiting queue. The waiting queue is a first-in-first-out (FIFO) or first-come-first-serve (FCFS) queue. We have provided an array-list based queue implementation in `pylistqueue.py`. You need not modify this class.

When you use the waiting queue, the nodes in the waiting queue should be Customer objects.

Store People

Your program needs two classes of people, `Customer` and `Cashier` that inherit from the given superclass `StorePeople`. You must determine what information and functionality each sub-class needs

to store. You may alter the superclass, if you determine your design would benefit. Your Cashiers should initialize to being available.

☑ ~~*In* `simpeople.py`*, create the Customer and Cashier sub-classes.*~~

In our simulation, we will have *n* cashiers, defined as the StoreSimulation attribute cashiers and initialized in the constructor as an array filled with Cashier objects. Note our use of Array, as we have a fixed number of Cashier's during the simulation.

Types of events and their handling

One center piece of a discrete event driven simulation is to define the various types of events.

`Event class`

The `Event` class is defined in `simulation.py`. It has class attribute constants for event type (explained below). Recall, to access class attributes (as opposed to member attributes), we use

```
Event.ARRIVAL
```

The Event class' member attributes are:

- `time`: integer– the time step the event occurs
- `type`: integer– constant (see below); identifies the event
- `who`: StorePerson, customer or cashier associated with the event

Our project will use the following two events:

`ARRIVAL` `event`

We initialize our simulation with a customer arriving at time 0. From there, the frequency of customer arrivals will be based on a discrete uniform distribution statistical model; i.e., *we will determine the time between two customer arrivals by generating a random number between* `1 and max_inter_arrival_time.` The program should keep a counter of the total customers arrived.

When a customer arrives (i.e., an `ARRIVAL` event is dequeued from the event queue), if no cashier is free, the customer joins the single waiting queue. If any cashier is free, the customer starts to receive service immediately (see section below). During this time, your program should also generate the next `ARRIVAL` event. If the current customer arrives at time *t*, the next customer should

arrive at time `t + random.randi` (using the arrival random generator attribute of the `StoreSimulation` class).

`DEPARTURE event:`

When a customer finishes the service at the cashier register (i.e., a `DEPARTURE` event is dequeued from the event queue), the simulation program should increment the total number of customers who finished the service with the store. Since this cashier is now free, the program checks to see if anyone is waiting in the customer waiting queue. If someone is waiting, take that customer off the queue and start the service immediately. If no one is in the waiting queue, the cashier is set to available in whatever way works with your implementation.

Serving a Customer

To serve a customer, you should (1) set cashier as busy, (2) calculate waiting time between customer entry and beginning of service and add to simulation total waiting time and (3) generate a `DEPARTURE` event and add to the event queue. The time of the `DEPARTURE` event will be determined by the customer service time, calculated as the

```
current time + a random number
```

from 1 to `max_service_time` using the service time random generator.

Statistics to be collected

The following statistics are to be collected.

1. Average waiting time in the cashier queue: This value is the total waiting time divided by the total number of customers who receive the service.
2. Total number of customers who completed the service and total number of customers who are left in the cashier waiting queue when the simulation finishes..
3. At the end of the simulation, also print such statistics as the total number of cashiers, total simulation time, and the total number of customers who arrived at the store.

File Overview

Here is a description for each of the files given.

- **`main.py`** – This is where you will ultimately run your simulation from and create incremental tests for your other code.

- **`array204.py`** and **`pylistqueue.py`** – These are the two files you'd use to create various queues in the simulation. You will need two types of queues in a simulation, FIFO queue(s) are used as the waiting queue for the customers, and a priority queue based on the event time is used to maintain the events in the event queue. You should not change any of these two files.
- **`linkedpriorityq.py`** – This file contains a collection of skeleton code for various methods used in a linked list implementation of a priority queue. You will need to implement most of these methods.
- **`simulation.py`** – The file contains a skeleton of class StoreSimulation. A complete constructor is given so you can see the attributes and their initial values for the class. A print_results() and the main() methods are also given. The print_results() method shows what statistics should be printed. The main() method shows how the program would run. You can tell from the main() method that **you will need to develop a run() method** for the simulation that runs the simulation logic.
- **`simpeople.py`** – This file contains a skeleton for the three classes you are asked to develop. A **StorePeople** class should be the superclass for the two subclasses, a **Customer** class and a **Cashier** class..

That's it. For Phase 1, you should not have to add new files, but put your implementation of the simulation in these files.

Phase 2: Implementing the event priority queue using a heap

In phase two, you are asked to use heap as the priority queue. A separate Repl stating Phase 2 will be used. Please do not change Phase 1 Repl after submission for Phase 2.

In the Phase 2 replit:

- ☑ ~~copy all files *except* `linkedpriorityq.py` and `main.py` from Phase 1~~
- ☑ ~~change one import statement in simulation.py~~

`# simulation.py, Phase 1`
 	
`from linkedpriorityq import PriorityQueue`
	`# simulation.py, Phase 2`
	
	`from heappriorityq import PriorityQueue`

- ☐ complete the priority queue implementation in `heappriorityq.py`

After changing the import statements, you should only need to edit heappriorityq.py to complete the implementation and main.py to write test code for your heap-based priority queue.

Note: This is one of the powers of object oriented programming and abstract data types. We can change the underlying implementation of our priority queue without having to change the code that utilizes it (e.g., the code in simulation).

4 Test Your Programs and Strategies for Development

Start your program with simple settings and easy to control tests. For example, define the customers, cashiers classes first and test them before using them in the simulation program. Write the tests in `main.py.`

When writing the simulation, begin by writing out the steps in comments. Work incrementally. For example. First write code to added the first Customer `ARRIVAL` (before the while loop). Run that code and print out information to insure it world. Then repeat the process after implementing just the logic for `ARRIVAL` events when a cashier is free. Then extend that to cover when there is not an available cashier. Then work on `END` events. Then finally add lines to collect needed statistics.

`debug variable`

Print statements throughout the program can help debug and check correctness. To make it easier to switch between printing debug statements and printing clean output, you can use a conditional and boolean variable.

For example, in a highly simplified example, we can use the debug attribute of the `StoreSimulation` class to create debug print statements in the run method.

`class StoreSimulation:`
`  def __init__(self, …):`
`  **self.debug = True**`
`  …`
`  def run(self):`
`  if self.debug:`
`  print("Debug inside run")`
`  print("Standard output")`
	`Debug inside run`
	
	`Standard output`

`class StoreSimulation:`
`  def __init__(self, …):`
`  **self.debug = False**`
`  …`
`  def run(self):`
`  if self.debug:`
`  print("Debug inside run")`
`  print("Standard output")`
	`Standard output`

The following is a snapshot of possible debug statements that may be useful to add to your code.. (Events between time 2 and 31 are not shown because of limited space).

```
Simualtion starts ...
Time  0 : Customer  0  arrived.
Time  0 : Customer  0  service starts by cashier  0
Time  2 : Customer  1  arrived.
Time  2 : Customer  1  service starts by cashier  1
Time  2 : Cashier  0  stopped serving customer  0 .
Time  4 : Customer  2  arrived.
Time  4 : Customer  2  service starts by cashier  0
...
Time  31 : Cashier  1  stopped serving customer  17 .
Time  31 : Customer  19  service starts by cashier  1
====== Simulation Statistics =======
number of cashier :  2
totalSimTime:  30
interarrivalTime:  2
cashier service time:  5

Number of customers served =  20
Number of customers remaining in line = 0
The average wait time was 0.30 minutes.
====================================
Simulation ends ...
```

Figure 4: Snapshot of running the simulation.

When your program works correctly, you can turn off the debugging mode and print just the summary of the simulation (the last block of the statistics in Figure 4.)

You are asked to test your program with the following two sets of parameters for Phase 1. You can try other configurations, but submit the results for these two.

1. `Number of cashiers 2, maximum customer inter-arrival time 2, maximum cashier service time 5, simulation time 1000.`
2. `Number of cashiers 5, maximum customer inter-arrival time 2, maximum cashier service time 14, simulation time 3000.`

5 Submission

Submit each phase separately to Repl.it which should include all program files (in order to run the simulation) and one *README.txt* file for each phase.

`README.txt`

In the *`README.txt`* file, include

- ☐ the project number, project phase and your name in the *README.txt* files.
- ☐ the result of sample runs,
- ☐ explain your program briefly, including any wins or challenges,
- ☐ Any extraordinary elements, creativity or innovation if you are aiming to receive those points.

6 Grading Criteria

While grading each phase, we will look for the quality of the program from various aspects as well as the correctness. Here are the grading criteria.

1. Functionality: 65%

   User-friendly and well-formatted input and output (10%)

   Correctness of computation (25%)

   Program satisfies the problem specifications (25%)

Incremental testing demonstrated through main.py tests and debug print statements (5%)

2. Organization and style: 15%
    a. Program is well-designed and follows an object-oriented design approach using proper class and inheritance structures. (15%)
    b. Good python style: e.g., variables and functions have descriptive names following Python convention (2%),
    c. Program contains sufficient amount of comments explaining the functionality major blocks of code; docstrings at the top of each method describing its parameters, their data types/class, a brief description; each code file has comments on the top including student name, date and file description (10%)
    d. Readme.txt (3%)
3. Extraordinary elements, creativity and innovation: 10%

This includes anything beyond the provided specification that makes your program stand out.