Task 5: Traffic Control Discrete Event Simulation

Description

This task investigates the third main type of simulation: discrete event simulation. The continuous time-stepped simulations in Tasks 1, 2, and 3 kept a running clock that incremented at a fixed rate. They updated the state of each entity at each moment in time. (Technically these are discrete steps because time does not exist between them, but do not confuse this use of discrete with the one in this task. They are similar concepts, but the simulation types are quite different.) This type of simulation is very useful for physical entities moving through space and time. The stochastic simulation in Task 4 was a combination of time-stepped and random generation of errors. (To a much lesser degree, so was Task 1 with the manual shifting performance.) It does not need to combine with a time-stepped simulation, as we saw in the Lecture 4 example of approximating π by randomly selecting positions within overlapping geometric regions.

In all the previous tasks, the emphasis was on the *transition* between states. For example, the dragster moved from the start to the end of the race. The start and end existed only to define the limits that the dragster needed to traverse. Similarly, the artillery shell fired and landed, but the true interest was on the flyout. In this task, the emphasis is on the *states* themselves, not the transitions between them. It models traffic at a simple intersection, but the physicality of the cars (i.e., moving through space and time) plays no role. The cars magically appear and disappear at the appropriate time, and we do not care how these states came to be or where they go once they are done. It is somewhat like my industry example from Lecture 38 with the Phalanx close-in weapons system that operated faster than the computer could simulate the extreme rate of fire. Almost all hits magically appeared at the target without actually flying out. Only representative sample flyouts were necessary to establish the statistical and probabilistic bounds to make the rest materialize appropriately.

The basis of this task is a dumb traffic light, which controls traffic in one direction only. It alternates at a fixed rate that we set in advance between red and green lights. For simplicity, there is no yellow. Cars that arrive when the light is red queue up and depart when it turns green. Cars that arrive during a green light pass right through unimpeded. (Although they do not queue up per se, we still record their arrival the same way for analysis.) The cars do not interact because they do not actually exist. Their presence is merely an accounting trick. This type of simulation is extremely efficient because it models only the arrivals and departures as instantaneous events. No intermediate actions occur, which means no processing time is wasted in updating the system when nothing is happening. Such simulations are extremely common for process flows like checkout and assembly lines. All computer programs are state machines.

The objective is to implement the simulation that drives the model, collects the data, and reports it. This time the analysis code gets more emphasis, but for practicality, you will not implement it as I had originally intended (because the analysis in Task 3 caused a lot of trouble). The model is provided as follows:

- `TrafficQueue` is a single traffic light and presumed approach area where cars line up to wait on red. Again, we do not model the cars, so the only thing we need to maintain is how many cars there are and when each arrived; i.e., in a ledger, which is `AnalysisGroup` below. For all practical purposes, they are stopped on top of each other at the first position before the light. There is no acceleration or deceleration or response time to account for. Collisions are not possible. There is no concept of lanes.

- `Event` is an entry in `TrafficQueue`. It represents the arrival of a car or the transition of the light from one state to the other; i.e., red to green or green to red.

- `AnalysisGroup` is a data structure for collecting when events occurred. See below for details.

The simulation in `Simulation` is your responsibility. You need to set up the traffic light, fill the queue with cars, and service them by holding them at a red light and discarding them at a green. The code skeleton includes pseudocode to guide you.

The statistics to report are collected and generated in `AnalysisGroup`. It is nothing more than a list of event times. Unlike `TrafficQueue`, however, it is not a queue. There are also as many instances of this class as necessary to collect data on different conditions in the system; i.e., cars at the current red light or green light, at all red or green lights, or all cars having visited the intersection. The same statistics are always calculated. The significance (or even validity) of the results depends on what is added to the list and what the questions are that the results address. This approach allows us to collect data for any number of questions simultaneously. It also allows for more complex and interactive systems like multiple traffic lights, but we have only one.

`AnalysisGroup` was originally intended to be your responsibility, but the complexity was a little too much. The original documentation is provided here in blue for reference, but you do not need to touch this class now. Note that it is not particularly elegant. It was supposed to be implemented in terms of coding you know from CS 210, 211, and 300.

Specifications

The Javadoc specifies `Simulation`, which you must complete. Do not modify the other classes, except if you need to output data for analysis. The following, along with Lectures 39 and onward, provide a description of how they work.

Time is always in seconds. Report all times to one decimal place.

You may assume that all input will be valid. Error checking is not required. Some documentation indicates error conditions, but these are only for clarification on how to use the code properly. It is very bad style to omit the defense mechanisms in any code, but we are being sloppy to reduce the workload. However, it does demonstrate that ordinary programming errors can mess up the results of a simulation. In other words, unless it crashes, it will produce numbers. Whether these numbers are meaningful with respect to the problem is a critical question. The same consideration holds for testing in software engineering. Refer to the lectures on verification and validation for more examples.

The nature of this task makes the documentation difficult to follow. You are expected to ask questions. We have a little extra time to work on this one.

`AnalysisGroup`

Instantiate a group with an identifier and initial time. The identifier is used only for reporting in the output. It is really easy to confuse the results without it. One of the hardest things about simulation and software testing is collecting and processing the test results in an organized, disciplined manner. The initial time is when the group starts recording events. This value depends on how the analysis group is being used. There are five groups used in `Simulation` (use these identifiers):

- `all` contains the arrival time of every car that arrived at the light from the start of the simulation (0 seconds) to the end (300 seconds). It summarizes the overall performance of the system.
- `on_green:all` is the same as `all`, except it contains only the green cars.
- `on_red:all` is the same as `all`, except it contains only the red cars.

- `on_green:`*n* contains all cars that arrived on one green cycle (or turn) of the light. Each cycle is independent. *n* is the cycle number, starting at 1.
- `on_red:`*n* is the same as `on_green:`*n*, except for red cycles.

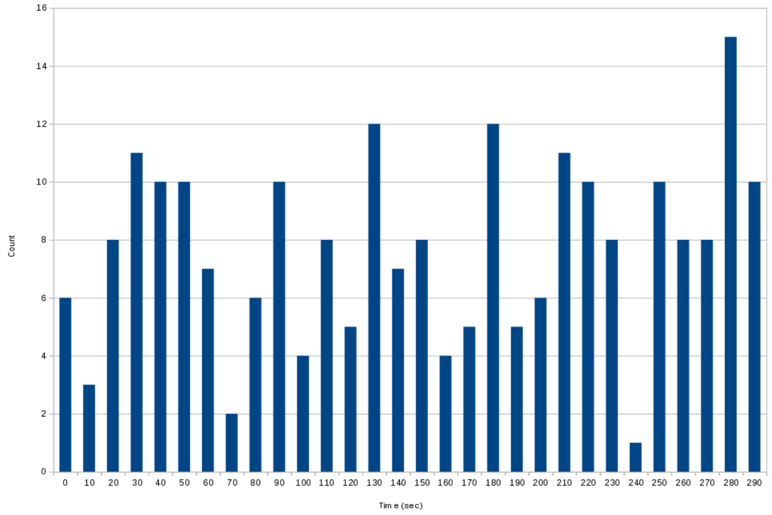
The initial time for the `on_green:`*n* and `on_red:`*n* groups is whenever the previous group ended since they alternate. For example, if 10 to 20 seconds was a green state for turn 1 and 20 to 30 a red state for turn 2, then `on_green:1` starts at 10 and ends at 20, and `on_red:2` starts at 20 and ends at 30 (when `on_green:3` in turn would start). At the end of each group, these statistics summarize what happened:

- `getEventCount()` returns the number of car arrival events.
- `calculateWaitTimeMin()` returns the minimum time a car had to wait; i.e., the last car arrived last and waited the shortest.
- `calculateWaitTimeMax()` returns the maximum time a car had to wait; i.e., the first car arrived first and waited the longest.
- `calculateWaitTimeTotal()` returns the total time all cars waited.
- `calculateWaitTimeAverage()` returns the average time that all cars waited. It is the total time divided by the number of cars.
- `calculateWaitTimeDeviation()` returns the *sample* standard deviation of the time that all cars waited. (Be careful: this is the n–1 version of the formula.)
- `calculateCarsPerSecond()` returns the number of cars per second serviced. It is the number of cars divided by the time span of the group (from start to end).

Most of these statistics depend on knowing the end time. Be sure to first call the `finalizeInterval()` method after the interval is complete (e.g., `on_green:1`). It outputs the statistics to standard output. Likewise, for intervals that accumulate over multiple turns, call `addEventSeparator()`. It converts the arrival times during this subinterval to wait times with respect to its end time. For example, the interval `on_red:all` from the start to the end of the simulation has a subinterval for each time the light turned red. The wait times in the subinterval are dependent on when the subinterval ended, not the interval itself. If you omit these intermediate calls, the wait times will be huge because they are calculated as `(300 - arrivalTime)`, not `(endOfSubintervalTime - arrivalTime)`. Work through the text output to be sure your results make sense.

All statistics are valid numbers, but not all make sense with respect to the data collected in their group. For example, wait times are meaningless for `on_green` groups. Likewise, cars on green skew the wait times in the `all` group. It is your job as an analyst (or tester in software engineering) to ignore meaningless data.

In addition to the standard deviation, we are also generating a histogram of the arrivals. The method `generateHistogram()` divides the interval into buckets of each time. The last bucket may not divide evenly into the interval. Each bucket contains the number of arrivals during this time block; i.e., from the previous time up to but not including the next time. The output is in comma-delimited format to be imported into Excel and plotted as a vertical bar chart: *`bucket_start_time,count`*. The histogram for analysis group `all` in `example.txt` is:



Simulation

Implement the solution as indicated in the pseudocode. See `example.txt` for the output from the initial configuration in `doTest()`, which is implemented for you.

We are running the simulation for 300 seconds. Remember, this is no explicit clock, so what we are really doing is generating random cars over an interval of 300 seconds. Once there are no more cars or lights, the simulation ends. We assume the light is initially green. There is no state event at time 0 to indicate this.

In `populateCars()` calculate the time between car arrivals with this random exponential function:

```
Math.log(1 - random.nextDouble()) / -lambda
```

Experiment

For each test, generate and save the raw text output (including histogram entries) and make an Excel histogram plot.

Test 1

Generate the analysis results from the initial test given in `doTest()`: `lambda`=0.75 and a red light every 30 seconds starting at 30, then green every 30 seconds; use a histogram bucket size of 10 seconds. Your initial attempt should match `example.txt`, but for the results of this test, use random seed 2.

Test 2

Rerun Test 1 with `lambda`=0.5.

Test 3

Rerun Test 1 with `lambda`=5.0.

Test 4

Rerun Test 1 with green lights lasting 45 seconds and red 30.

Test 5

Rerun Test 1 with green lights lasting 30 seconds and red 45.

Test 6

Rerun Test 1 with green lights lasting 10 seconds and red 10. Even if the results are positive, why is this approach troublesome in the real world?

Test 7: Grad Students Only

Using `lambda=0.75` and assuming red and green lights are the same length, what is the longest red light that does not hold back more than 10 cars ever? Hint: the statistics output from `AnalysisGroup` does not explicitly report this result for you. You must inspect all the values of `numCars` behind red lights yourself to find whether any red light exceeded 10. You are free to write support code, if necessary.

Test 8

Rerun Test 3 but use a bucket size of 15 seconds.

Test 9

Rerun Test 3 but use a bucket size of 5 second.

Deliverables

Submit the following:

- Your Java code for `Simulation` only. No other code should need changing.
- The eight (or nine) output files called `output_`*n*`.txt`, where *n* is the test number.
- A nicely formatted PDF document that appropriately indicates the results of each test. Present and possibly summarize the analysis results for each.