İSTANBUL TECHNICAL UNIVERSITY
Department of Computer Engineering
BLG443E – Discrete Event Simulation – Spring 2019
Assignment 2 – Simulation with a general purpose language.

ASSIGNMENT 2

Assignment description

You are again modelling the cabbage expiry scenario, but this time you will use the **Event-List Processing** or **Two-Phase** approach and software engineering.

Your job is to simulate the shop described in the assignment 1 scenario, and again do the optimal stock size experiments (decide on the optimal stock level of cabbages to keep the number of expired cabbages to a minimum while maintaining the number of customers unable to find a cabbage in stock to 3 in 1000 or less).

Recall that: cabbages go rotten in 7-12 days, are delivered in 1-15 days, and shoppers looking to buy cabbage come every 0-3 days (you may assume these are uniformly distributed).

Experiments

Optimal stock size experiment: You will use this new system to conduct the same experiment you did in the first assignment: Determine the optimal number of stock to be kept on the shelf when using the unchanged model by running an automated experiment to determine the optimal level. The optimal level is the one that always keeps the number of customers refused cabbages less than 3 in 1000 and minimises the number of expired cabbages. Use as many trials and as many customers as is feasible considering your implementation. Check each configuration and automatically choose the configuration that best fits the criteria. *Which is the optimal configuration?*

Programming

Simulation Language. You must create your simulation and analysis in of the domain using one of the languages of assignment 1 (Python 3 or Cling/C++17), but in contrast to assignment 1 you are required to use generic simulation programming methods, and in particular *Event-List Processing* or *Two-Phase* approaches.

Simulation Methods. You are required to:

1. Use the *Event-List Processing* approach to discrete event simulation.
2. Use a software engineering approach where a user can set up a *process flow* and then simulate entities passing through the flow.

In the case of the cabbage shop, the entities passing through the system will be cabbages and customers. The blocks in the process flow will be arrival of an entity (an entity might be a cabbages or a customer – such blocks are customarily called “generate” or “source” blocks because they generate new entities), the passage of time (the process of cabbage shelf-life or re-ordering process), entities leaving the system (cabbages going rotten or being bought, customers leaving after buying a cabbage, or leaving unhappy – such blocks are customarily called “terminate” blocks), or interactions (customers buying cabbages, or attempting to).

Experiments should be in a Jupyter notebook with generic simulation code (that is, code that does not depend on the particular scenario being simulated) in .py or C++ files. In your notebook, you should be able to make use of your simulation something like the below listed code (**listing 1**, which simulates the arrival and leaving of customers).

⚠

In this code, **des is a module to be implemented by YOU**. That is the main aim of this assignment (apart from conducting the required experiments): implementing a class that can be used as below.

Listing 1

```
import des

cust_entry= des.GenerateEntityUniformDistribution(low=0,high=3)
cust_counter = des.EntityCounter()
cust_leave = des.TerminateEntity()

cust_entry.set_target(cust_counter)
cust_counter.set_target(cust_leave)

simulation = des.Simulation([cust_entry])
simulation.run(stop_after=(cust_leave,10)) #stop when 10 customers have left.
print(cust_counter.total_count(),"customers were simulated.")
```

ⓘ

The above code is in Python but it has been written in a simple way so that it could be written similarly in C++ - it is suggested, if you use C++, to use *smart pointers* to objects, which in C++17 can be made easier through the use of the *auto* keyword).

An alternative attempt, encompassing just the stocking of cabbages on their shelves and their tendency to be re-ordered might be as follows:

Listing 2

```
import des

first_cabbages = des.GenerateAtStart(num=3) # for 3 cabbages in stock
cabbages_on_shelf = des.AdvanceTimeUniformDistribution(low=7,high=12)
cabbage_rotten_cntr = des.EntityCounter()
cabbage_reorder_proc = des.AdvanceTimeUniformDistribution(low=1,high=15)

first_cabbages.set_target(cabbages_on_shelf)
cabbages_on_shelf.set_target(cabbage_rotten_cntr)
cabbage_rotten_cntr.set_target(cabbage_reorder_proc)
```

```
cabbage_reorder_proc.set_target(cabbages_to_stock)

simulation = des.Simulation([cust_entry,first_cabbages])
simulation.run(stop_after=(cabbage_rotten_cntr,10)) #stop after 10 go rotten
print(cabbage_rotten_cntr.total_count(),"cabbages went rotten.")
```

😸

If you have a better idea for a software architecture for this problem, feel free to use that one instead – we would be happy to see it.

A more complete example, encompassing both cabbages and customers and their interactions might be as follows.

Listing 3

```
import des

first_cabbages = des.GenerateAtStart(num=3) # for 3 cabbages in stock
cabbages_on_shelf = des.AdvanceTimeUniformDistribution(low=7,high=12)
cabbage_rotten_cntr = des.EntityCounter()
cabbage_reorder_proc = des.AdvanceTimeUniformDistribution(low=1,high=15)

first_cabbages.set_target(cabbages_on_shelf)
cabbages_on_shelf.set_target(cabbage_rotten_cntr)
cabbage_rotten_cntr.set_target(cabbage_reorder_proc)
cabbage_reorder_proc.set_target(cabbages_to_stock)

cust_entry = des.GenerateEntityUniformDistribution(low=0,high=3)
get_cabbage = des.DisplaceEntity(from=cabbages_on_shelf,to=cabbage_reorder_proc)
cust_leave_happy_cntr = des.EntityCounter()
cust_leave_unhappy_cntr = des.EntityCounter()
cust_leave_happy = des.TerminateEntity()
cust_leave_unhappy = des.TerminateEntity()
get_cabbage.add_transition(cust_leave_happy_cntr,alternative=cust_leave_unhappy_cntr)
cust_leave_happy_cntr.set_target(cust_leave_happy)
cust_leave_unhappy_cntr.set_target(cust_leave_unhappy)
simulation = des.Simulation([cust_entry,first_cabbages])

simulation.run(stop_after=(cust_leave_happy,10)) #stop when 10 customers leave happy.
print(cust_leave_happy.count(),"customers went home happy with cabbages.")
print(cust_leave_unhappy.count(),"customers went home unhappy without cabbages.")
print(cabbage_rotten_cntr.count(),"cabbages sadly went wrotten.")
```

🗝

Try drawing a ***process-flow diagram*** from the provided code. It will help you to understand what is going on. Advanced students may wish to try generating these diagrams programmatically.

Write your simulation from scratch. Suggested steps are as follows:

1. Create a future-event-list and populate it with the first customer arrival.
2. Write the simulation loop in which you pop the customer arrival and put the arrival of the next customer on there (this is called “bootstrapping” - you cannot know ahead of time in the general case how many customers there will be so you only queue up the next customer when one arrives), being sure to update the clock when the customer arrives.
3. Expand the simulation loop so that you keep track of how many customers have arrived in total.
4. Refactor your code along the lines of that in **listing 1** to simulate customer arrivals so that you can build a simulation of customer arrivals in the way suggested there.
5. Add new blocks one by one till you can fully specify the simulation as suggested in **listing 1**.
6. Add new blocks one by one till you can fully specify the simulation as suggested in **listing 2**.
7. Add new blocks one by one till you can fully specify the simulation as suggested in **listing 3**.
8. Run the required experiments in a Jupyter notebook (trying different stock sizes to see which one minimises cabbage rot while keeping the number of refused customers below 3 in 1000) and report your results graphically. Compare verbally them to your results from assignment 1.
9. Do any advanced implementations or analyses later in your notebook (for example, generating process flow diagrams automatically, improving readability of model code using language-specific features, doing an automatic parameter search, etc.).

Submission policy:

- Only electronic submissions through **Ninova** will be accepted.
- **Late** submissions or submissions submitted otherwise than according to instructions will not be accepted.
- Submit:
  - All .py *or* .cpp files containing your generic simulation code (in a **ZIP** file).
  - A Jupyter notebook running the cabbage-stock experiment using your simulation code (in **ipynb** format), showing graphically the results.
- You are not allowed to use special purpose simulation software and modelling specific libraries like GPSS, Arena, SimPy, Simula simulation package, SSF, and so forth. Do not use platform-specific or non-standard (not built-in) libraries apart from the ones mentioned here or in assignment 1 (Python: Python Standard Library, Matplotlib, numpy, scipy. C++: C++ Standard Library, Boost/random, xplot).
- Academic dishonesty (this includes cheating, plagiarism, direct copying of code) is unacceptable.

Criteria:

The main criteria for a successful assignment are:

- **Correctness of generic simulation code.**
- **Correct simulation logic (cabbage arrival, expiry, re-ordering, customer arrival and leaving, interactions).**
- Correct collection of statistics.
- Correct experimentation code.
- Clear of code.
- Correct output.
- Correct histograms.
- Appropriate use of experimentation.
- Clear presentation of experimental results.
- Clear recommendations.
- Justifications referring to the results.
- Cool other things that you have done, justified, and documented carefully.