Lab 2 – Modelling

Candidate number: 10026, Team member candidate number: 10093

The code can be found https://github.com/nemisis84/PYSE. There have been changes to the models, and new versions of the diagrams in lab 1 are added as appendixes. The changes were made based in the feedback from lab 1, and practical considerations when coding. For instance, there were added a shelf_available list which ensures that only one employee at the time work at one shelf.

II.A.1

This task is solved using np.random.choice(5, 7) [1], which returns a list of 7 integers randomly drawn from a uniformly distribution between 0 and 5. The function can be found in main.py in the Customer class as a private method. This only creates the shopping as the generator for the customer itself are solved in II.B.1

II.A.2

The MOS-score can be found in main.py in the Customer class.

II.B.1

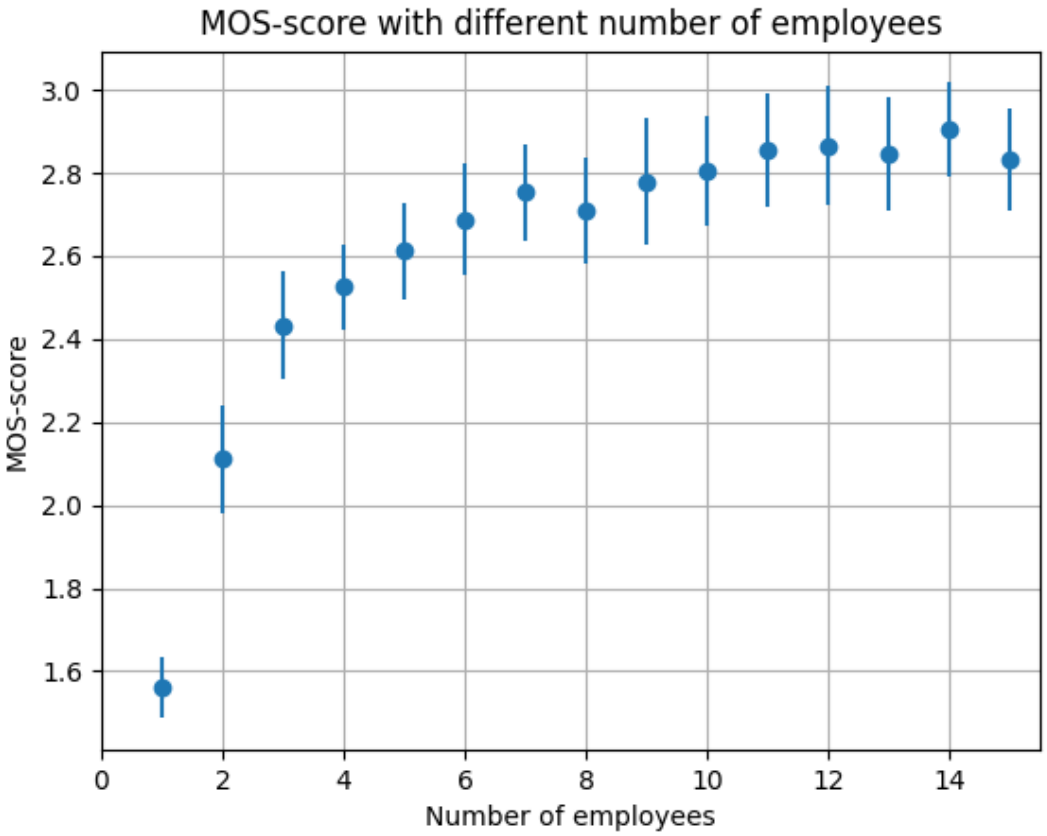
The customer generator can be found in main.py, and are called generator_customers(). It constantly creates customers until the end of the simulation time. The simulation of customers method is called customer_shopping and lays within the customer class.

II.B.2

The employee generator can be found in main.py, and are called generator_employees(). It Creates n employees. The employees refill the shelves using the method refill in the employee class. To ensure that only one employee is at the same shelf at the same time, there are a Boolean list called shelf_available which becomes False as long as an employee is filling up a shelf. The n employees will walk around the store and refill until the end of the simulation. This is opposite of the customers, which comes and leaves, but is constantly generated.

II.C.1

30 simulations were run for each employee count. From this, the mean and standard deviation was calculated. Here are the results in a point plot with error bars.



The model is highly simplified in many areas so the results may be somewhat imprecise. Without knowing the costs of an employee and the befit of a higher MOS-score, there are hard to give a recommended employee count. However, something in the range of 4 to 11 employee makes sense based on the plot above. Fewer than 4 employees will lead to unhappy customers, while more than 11 doesn't impact the MOS-score significantly.

Something to be considering is the timing of the timeouts in the code. For instance, when an employee refills, the timeout occurs before the refilling. However, you could say that the products will be available while the employee refills. Whether the timeout is before or after the refilling will significantly impact the MOS-score.

II.D.1

The MOS-score doesn't consider whether you successfully panted or not. This should be considered. The score does not account for the time spent in the store or time in the store per item either. The total time in the store could affect how satisfied the customers are. However, all the time steps in the system are constant apart from the queue time. Therefore, there are no point of taking the total time in the store or time in the store per item and using only the queue time is sufficient.

The score does consider which percentage of items the customers get. This is a decent way to measure the satisfaction of the customers. However, could be events where a customer does not get many items, but returns a decent MOS-score because the customer has bought many items. Another solution here could be to just count the number of items not gotten.

Subjective consideration such as, overall experience in the store is hard to measure. However, no one likes to move in a crowded store. Therefore, there could be considered to add people_in_the_store_while_shopping. This indirectly goes into the queue time but is something that could measure the satisfaction of the customers while shopping.

II.D.2

My suggestion is a mean_queue_time variable. This variable keeps track over the mean queue time for the last n customers. n could for instance be something between 5 and 20. What n would be should decided by running the simulation with multiple n-values. The store will turn on counters if mean_queue_time > some_constant and turn counters off if mean_queue_time < some_constant. There would be multiple thresholds which controls whether 1,2,3,4, … n counters should be open. The downside with this method is that it is reactive. It will not take preventive measures as it will lead to some customers having to wait for a long time before it is opening a counter. This method could easily be added in the code as well as in the activity diagram:

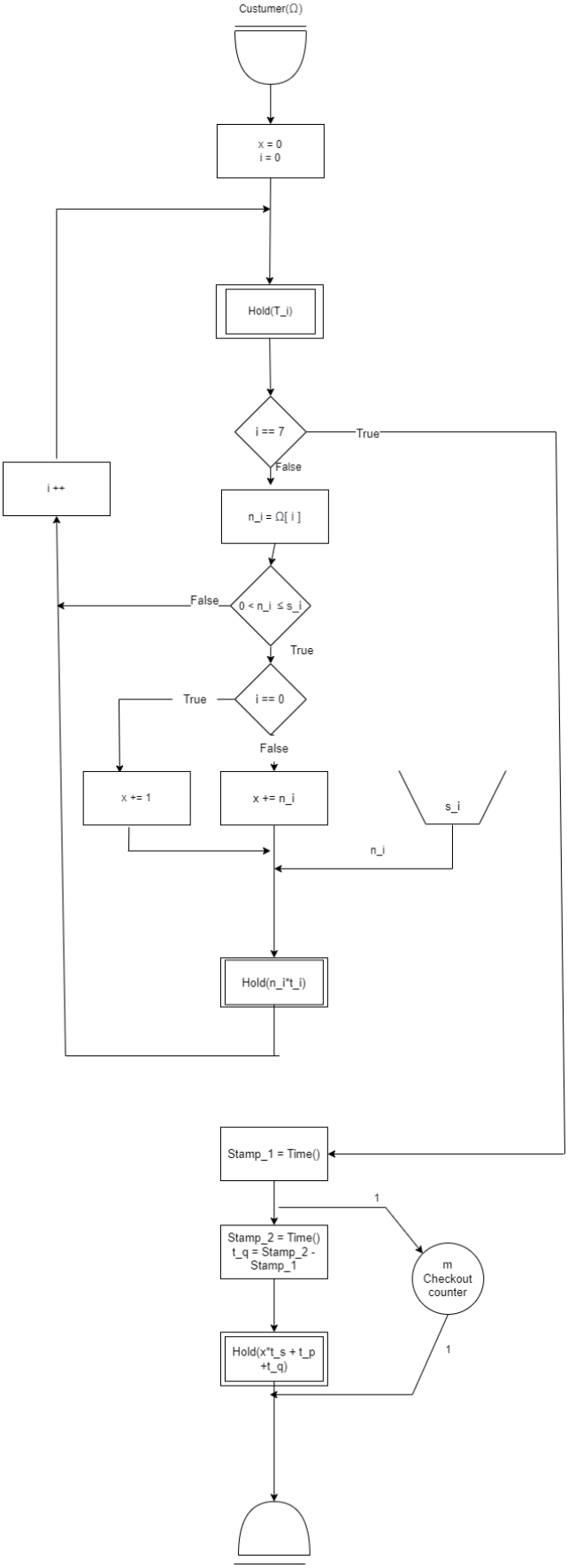
1. Create a list with length n, and a pointer.
2. Create lower and higher limits for mean queue time.
3. When a customer leaves the store replace the value at the pointer with it the queue time and increment the pointer.
4. Calculate the mean and check if it is above or under some of the limits.
5. If higher_limit< mean or mean<lower_limit →Add/remove counters accordingly

Another method could be a variable n_customers_in_the_store. Create a variable customers_per_counter = n_customers_in_the_store/counters. Then create a lower and higher limit for how many custumers there should be per counter. If customers_per_counter>higher limit add a counter, if customers_per_counter<lower limit remove a counter. This is more preventive than the other method. However, it doesn't consider the queue time. If the customers in general buy a small amount of items there might be unnecessary many counters open, and if the customers have buy a large amount of items there might be too few counters. This method could easily be added in the code as well as in the activity diagram:

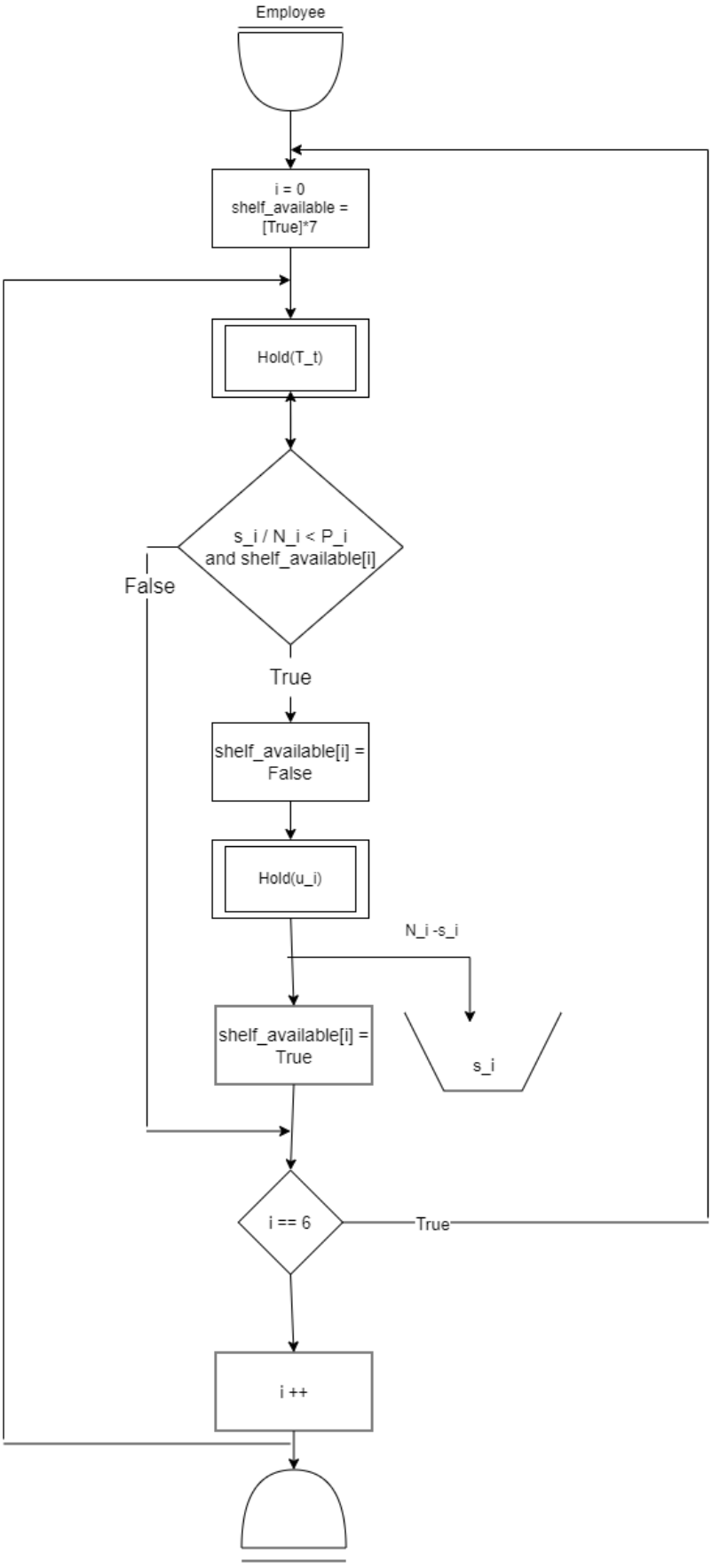
1. Create variables n_customers_in_the_store, counters
2. Create lower and higher limits for customers_per_counter = n_customers_in_the_store/counters
3. When any employee has finished its round around the store, check if higher_limit <customers_per_counter or customers_per_counter <lower_limit→Add/remove counter

I would recommend the second method.

Appendix A


Custumer(Ω)
x = 0
i = 0
Hold(T_i)
i == 7
True
False
i ++
n_i = Ω[i]
0 < n_i ≤ s_i
False
True
i == 0
True
False
x += 1
x += n_i
s_i
n_i
Hold(n_i*t_i)
Stamp_1 = Time()
1
Stamp_2 = Time()
t_q = Stamp_2 - Stamp_1
m
Checkout counter
1
Hold(x*t_s + t_p +t_q)

Appendix B


Employee
i = 0
shelf_available =
[True]*7
Hold(T_t)
s_i / N_i < P_i
and shelf_available[i]
False
True
shelf_available[i] =
False
Hold(u_i)
N_i -s_i
shelf_available[i] =
True
s_i
i == 6
True
i ++

References

[1] https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html
07.10.2022