Project Summary

In this project, we used the implementation of Queue to simulate the waiting lines in front of supermarket’s checkouts. Queue is a data structure that follows the rule of “first in, first out”. In this project, a doubly-linked Node-based structure (i.e. each Node will have a field of next and previous which respectively points to the next and previous Node in the Queue) is used to create the Queue, which has three basic operations. offer() adds an item to the end of the Queue, poll() removes and returns the first item of the Queue, and peek() returns the first item in the Queue.

In the base project, there are three types of customers, each of which use a different algorithm to choose a queue. Then I evaluated the time taken for each line-choosing method in larger simulations where 10k customers were added to the landscape and computed the average time taken and standard deviation for each method in the simulation.

Task Solutions

Customer

This is an abstract class that will be inherited by different type of customers that use different algorithm to choose checkout lines. It has the protected fields of the number of items, and the time steps taken before the customer leavers the store. The abstract method chooseLine will be override by its child class.

CheckoutAgent

The CheckoutAgent class has a field of a queue of Customers and two integer fields for the location of the checkout agent. The draw method draws a rectangle at the bottom of the window with a height proportional to the number of customers in the queue. Because the drawRect() method of java Graphics takes the coordinates passed in as parameters as the upper-left corner of the rectangle, so if the rectangles are drawn at the bottom of the window, the y-coordinates of the rectangle should be the this.y – the height, which is 10\*N.

PIC

Landscape

The Landscape class has fields for its width and heights, and an ArrayList of checkout agents in the landscape, and a LinkedList o all the customers who have left the line after all their items are checkedout.

RandomCustomer

The first class of customers implements the random line-choosing strategy. The initialized time step is 1 because the customer will pick one line randomly. It inherits the abstract method chooseLine from the Customer class, which generates a random integer that index the chosen line using the Random object.

Visualization testing RandomCustomer: when the sample used simulation is large enough, the random strategy should result in equal number of customers waiting at each checkout line. When the sample is smaller, the distribution will have more variations.

PickyCustomer

The PickyCustomer chooses the line that has the smallest number of customers in queue, therefore its initialized time step is equal to the number of checkout lines as it needs to evaluate the length of each line. I used a for loop in its chooseLine method to evaluate the number of customer in each line using getNumInQueue() and constantly update the line index if the length of a checkout line is shorter.

The visualization of PickyCustomer should result in an equal distribution of customers at each checkout line, regardless of population of the sample. In addition, each new customer added to the landscape will choose the next line of the line chosen by the previous customer, in cycle from index 0 to 5. Below is an image showing how 100 PickyCusomters are equally distributed among 5 checkout lines.

PIC

Pick2Customer

The Pick2Customer first randomly picks 2 lines and then chooses the shorter one. It spends two time step because it needs to evaluate the length of 2 lines. I first generated 2 integers that indexes the lines using the Random object; in order to make sure that two different lines are chosen, I used a while loop to repeat choosing the second index until it is different from the first index. The visualization creates a nearly equal distribution of customers among the checkouts, despite some small variations.

Simulation and Statistics of each strategy

The updateState method in CheckoutAgent class first increment the time for all customers in the Queue, and use the peek() method of Queue to look at the first customer and makes it to giveUpItem(). Once a customer has given up all its items, it will removed from the queue by calling poll() and added to the finished list of the landscape. The updateCheckouts() method in Landscape calls the updateState() method on every CheckoutAgent in the landscape.

Then I wrote the Simulation class for each type of the customers. I’ve run each simulation using different upper limit of the number of items that each customer could hold by passing in a command line argument, and adjusted the number to find the situations where the queue could balance. For example, when the maximum number of items a Random Customer could hold is 10, the checkout queue will increase very fast, i.e. the number of new customers joining the lines is more than the number of finished customers, as shown below.

When I adjusted boundary of number of items to 6, the lines are balanced, neither increases drastically nor decreases to 0, as shown below.

The strategy implemented by PickyCustomer has a higher tolerance for the boundary of the number of items. The simulation is balanced when the value is 8, as shown on the right. The left shows an imbalanced situation when the value is 10.

The simulation of Pick2Customer is balanced when the value is 9.

To compute the average and standard deviation of the time-to-leave for all Customers in each simulation, I wrote printFinishedCustomerStatstics(), which loops through every customer in the finished list and compute the average, and loop again to first compute the variance using Math.pow(), and compute the standard deviation using Math.sqrt(). Below is the statistics when I used 10,000 customers and run each simulation with the range of number of items that I got previously.

The PickyCustomer simulation has the most stable time-to-leave but the highest time average, probably due to high initialized time steps because they need to evaluate every line.

Extensions

New choosing line strategy: CountItemCustomer

This class of customer choose line based on the total number of items at each checkout line, so it will loop through all the customers at each line and call getTotalItem() and sum the results. Therefore, its initialized time step is equal to the total number of customers in the landscape when it is added. Its chooseLine method will return the index of the line with the smallest total number of items. this strategy has an even higher tolerance for the range of item numbers. For instance, when the value is 10, the lines are growing at a slower rate than PickyCustomers. The value for the lines to reach equilibrium is 9. Below is the recording showing the simulation with 10,000 customers.

I found that the height of lines will first increase slightly (but still in a controlled range), and then decrease for a period of time in the middle of simulation where the lengths of lines become really short, and eventually resume back to its beginning level.

This strategy has an overall average of 88.95 time steps and its standard deviation is 47.96. The average is high because of the high initialized time steps for a CountItemCustomer, which largely increases when more and more customers join the lines.

Customer Strategy Simulation

This simulation allows each customer to choose a random strategy from the four (Random, Picky, Pick2, Count Item) and mix the four types of customers in a landscape and compute the average and standard deviation of time steps for each category of customers. I first created an int field strategy in each customer class.

I added an ArrayList that holds four LinkedList, each of which holds the finished customers from each category. The location of the LinkedList in the ArrayList corresponds to the strategy field of each customer class. addToCategoirzedList method classifies the finished customers and add them to the finished\_categorized list. The updateStateCategorized in CheckoutAgent class will class the addToCategorizedList instead of the original addFinished method.

The statistics result will be printed through the printStrategyStats in Landscape class, which uses a nested for each loop to loop through the four LinkedList in finished\_categorized list and each customer in its respective LinkedList. The method will print out the class name of a customer class by calling getClass() and then its average and stand dev, as shown below.

The Customer Strategy Simulation will first for user’s input for the sample size and the upper range of the number of items. Then it will randomly creates customers from each type, add them to the queues and update the landscape. Below is the result of the overall performance of customers using different strategies.

The stats show that Pick2Cusomter has the shortest average time-to-leave and PickyCustomer has the smallest variations. CountItemCustomer has the highest average but not substantially higher than the other 3 classes, despite of the fact that it has a much higher initialized time step than the other three.

Thoughts: if we consider the initialized time step, i.e. time taken to evaluate the situation, as an intellectual effort, and differentiate it from the actual waiting time in the lines, CountItemCustomer should have the most efficient performance.

Improve visualization

I changed the color of the rectangles that represent the lines and simultaneously printout the number of customers in each line by using Graphics.drawString. All the changes are made in the draw method of CheckoutAgent.

Array-Based Queue

This is a generic Queue implemented using arrays. It’s similar to the generic array I wrote for project 4, but to implement Queue-specific methods, I changed the general add and remove methods to a way that it can only deal with adding at the end of the array and removing the first element. As tested in my main function, it works as the Node-based Queue.

Conclusion

In this project, I have practiced extensively the implementation of Queue and working with different classes that are associate with each other. I also practiced Node-based Queue and doubly-linked list. In addition, I practiced how to run simulation with a large sample and work with the data effectively.

Credits:

Prof. Layton

Prof. Maxwell’s lecture notes

Roujia Zhong

TA Michael Yu