Introduction

Running a business is particularly difficult because one is simultaneously concerned about costs as well as the profits obtained by operating such a business. As a consultant hired by a coffee shop startup, the main goal at hand is to determine the number of cashiers that the shop should hire to ensure that profits are maximized. The coffee shop is said to open at 6 am and closes at 10 pm, hence customers arriving exactly at the opening/ closing times as well as in between those times are eligible but not guaranteed to be served. Depending on the number of cashiers available, the choice is then made to serve the customers upon arrival or have them wait in line until there is an available cashier. The length of the line cannot exceed eight times the number of cashiers hired by the store; if the former is true the customers arriving will be turned away.

With these constraints in mind, the goal was to develop a simulation that would emulate the functioning of this coffee shop while keeping track of the profits as the number of cashiers that the store hires increases. Ultimately the number of cashiers that yields the highest net profit, would be the number of cashiers the coffee shop hires.

I hypothesize that as the number of cashiers is increased, the number of potential customers served will increase hence elevating profits of the Coffeeshop. This is because as more cashier become available, it will decrease the frequency for which customers wait in line, ultimately decreasing the number of customers that get turned away.

Approach

The initial design of my program consisted of four classes: Launcher/Controller, Coffee Shop, Customer, and Event. These classes will be discussed in detail in the following paragraphs.

Irwin Frimpong – October 14th, 2018 – CS 150 Project 1

The Launcher class is where the simulation of the Coffee Shop will take place. There is not much breadth to this class because the content that drives the simulation is performed in the Coffeeshop Class. The launcher creates an instance of the CoffeeShop class and runs the instance of CoffeeShop's run in the run method of the Launcher class.

The Customer class essential holds the metrics of each customer that is served in the Coffee Shop. Each customer has an arrival time, a wait time, depart time, and name as defined as instance variables at the top of the class. The Customer class includes getter and setter methods: `setDepartTime`, `setWaitTime`, `getWaitTime`.

The Event class, is one of the pivotal class in this project is responsible for dealing with the metrics associated with each event. Considering that there are two types of events, an arrival and a departure, constant int type variables `ARRIVAL` (which is set to 0) and `DEPARTURE (`set to 1) were created to differentiate between event types. This is imperative for the functioning of the simulation. Every event must have a type as discussed earlier, but they must also have arrival time, departure time, wait time, and name. These are taken care of by creating instances variables that will hold these values as their computed per event. Further, an instance of the customer is created that is passed the event arrival time and the name of the event. The idea was to have every instance of the customer associated with the same instance of the event, hence why the instance of the customer is passed into the constructor of the Event class, along with the arrival time, the type, and name of the event. Moreover, the Event class includes a `compareTo` method, getters and setters (`setTime`, `getTime`, `getName`, `getType`, `setType`), `departTime` and `waitTime` method.

The Coffee Shop is another essential class of this project, in that it, as mentioned earlier, houses the simulation. We'll first discuss the instance variables defined in the class and their significance. The `num_of_s` and `s_available` variable both refer to the number of cashiers hired by the store, however, the difference between the two is that `num_of_s` stores the number of cashiers specified by the user upon

running the simulation and `s_available` changes as the simulation runs to keep track of the available cashiers to serve customers. Variables `p`, `c`, `t` store the profit per customer, cost of staffing a cashier per day, and serving time respectively; these values are given in the input files which are then assigned to these instance variables after scanning. `num_of_served` is initiated to zero and this keeps track of the number of customers served and `overflow` which is also initialized to zero keeps track of the customers who are turned away. `Netprofit`, `totalcost` , `totalprofit`, `overflow_rate` keeps track of their respective values. An empty Priority Queue of events is initialized which would have the sole responsibility of storing the arrival times of customers (which is essentially the times given in the input file), in the Priority Queue while also ensuring that these arrival times are sorted.[1] Considering that the arrival times given in the input file was in the 24-hr format with AM and PM, the only plausible way to deal with time was to convert all the times into seconds. An empty Queue of events implementing a LinkedList is also initialized to serve the purpose of housing the events/customers that arrive when there are not available cashiers to service them.[2] A LinkedList implementation was imperative in that it ensures constant runtime for adding to the end of the Queue and removing from the front of the Queue as cashiers become available. In addition, an empty ArrayList of Customers is initialized which stores the instances of customers per event; Every instance of a customers stores the customer's metrics (arrival time, departure time, and wait time). The scanner is further used in the Coffee shop to read in the arrival times from the input file, while simultaneously converting the time to seconds and then adding to the PriorityQueue of events. This PriorityQueue would be used in the simulation. Moreover, the methods in CoffeeShop include `stringSplit`, `convertTime`, `addToPQ`, `getPQ`, `calculateNetProfit`, `calculateTotalProfit`, `calculateTotalCost`, `calculateOverFlow`, `maxWaitTime`, and `avgWaittime`. An in-depth explanation of

[1] API, J. (2018, June 23). Class PriorityQueue. Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

[2] API, J. (2018, June 23). Interface Queue. Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

how the simulation was implemented and the methods would be discussed in the Methods section of this report.

Methods

The Scanner CoffeeShop:

As mentioned earlier in Approach, the CoffeeShop class houses the simulation, but how does the simulation work? Lets first explore how we input data into the Priority Que and further process it in the simulation.

The very first request from the user is to specify the number of cashiers that should be employed in the simulation. This value is stored in the `num_of_s` variable mentioned earlier and would be used later in the simulation. The input files of customer arrival times always start with the first three lines corresponding to the p, c, t values of the store. To ensure that these three values are not included in the PrioiotyQueue of events, I employ a counter variable `line_count` which is initialized to zero, corresponding to the first line in the document. With the use of if statements and the `line_count` variable, the values on the first three lines are assigned to their corresponding variables in the program.

After the third line or when the value of `line_count` is greater than two is when the scanner has reached the line corresponding to the first arrival event and can further process it and the following lines as events. This is done by first creating a String variable `time` that holds the arrival time in 24-hr format "06:00:00 AM". To be able to convert this String values into seconds, the `stringSplit(time)` method is called and the String `time` is passed as a parameter. In this method the String is split by ":" to separate the hours, minutes, and seconds and then split again by a space to separate the "AM" or "PM" time indicators from the numerical time value. These values are stored in an array that is returned and set equal to a String array `time_a`. Instances of the customer class are created with the constructor having the parameter of arrival time and name. To get the arrival time the method

`convertTime(time_a)` is passed into the constructor of the customer. The `convertTime(time_a)` method converts the array of the 24-hr time format into seconds. The `time_a` array is passed as a parameter in the `convertTime` method; the zeroth index of the array corresponds to the hour and that is multiplied by a factor of 3600, the first index corresponds to the minute which is multiplied by a factor of 60, and the second index corresponds to seconds. Each of these computations are stored in int type variables `hours_to_sec`, `mins_to_sec`, and `sec` respectfully. These values are then added together and stored in a variable of int type `time_conv` which is returned by the method. An instance of Event is also created with constructor parameters of the instance of the customer, the arrival time in seconds (the `convertTime(time_a)` method is called), the type of event which is an arrival, and the name (which is analogous to the name of the customer). The event is then added to the priority queue and the program loops to the next iteration repeating the steps mentioned above until the scanner reaches a line in the input file that is null.

The Simulation:

The instance variables used in the simulation are `s_available` as discussed earlier, `current_time` (used to keep track of customer/event times to calculate customer metrics), a new instance of Event `s`, `wait_time`, and `depart_time`. Since we are actively removing events from the priority queue, the simulation starts off by checking if the PriorityQueue `shop` is not empty, if that case is true, we then go ahead to set the event we remove from the PriortiyQueue to the instance of Event that was created.[3]

Since the store opens at 6:00 AM and closes at 10:00 PM, these times were converted into seconds and used in the if statement to check whether the event/ customer arrives within the times in which the store operates. The `current_time` is set the event's time which in this instance is the event's arrival time. The next check is done to see if the event is an arrival type; if that is true it is then

[3] Weiss, Mark Allen. Data Structures & Problem Solving Using Java. Pearson Education, 2010. Chapter 13: Simulation

imperative to check whether the number of available cashiers is greater than zero so that way that customer can be served. If all cases are true, we decrement the number of cashiers and we increase the number of customers served. Once the customers are served, the `depart_time` is calculated by calling the `departTime(current_time,t)` method with parameters `current_time` and `t`. Since we have current time currently set the arrival time of the event, the `departTime` method calculates the departure time of a customer by adding `t` (time served) to the `current_time` and setting that equal to the `depart_time` variable. This variable is passed to the `setTime` method of the instance of event which changes the time associated with the event to its departure time. The same departure time is passed to the `setDepartTime(depart_time)` method of the customer for the instance of customer to store its departure time. Since customers who are automatically served because there are cashiers available, don't have any wait time, their wait time is set to zero and the `setWaitTime(wait_time)` method with parameter `wait_time` assigns that wait time values to the instance of customer. The customers are then added to the ArrayList `waittime` of type customers, which would be used later in the program to calculate maximum wait time and average wait time. The Event type is set to a departure using the `setType(Event.DEPARTURE)` method of the Event class and is added back into the priority using the `addToPQ(s)` method of CoffeeShop.

Contrary to the case of having available cashiers, customers will then have to wait in line which is essentially the Queue `line` created in the CoffeeShop class. Before they are added to the Queue, the Queue size has to be verified to be less than 8*`num_of_s` , if so then they are added. This is where the distinction between num_of_s and s_available becomes important because if one variable was used simultaneously to keep track of the number of cashiers hired and those available to service customers, the condition statement for customers to be added to the Queue would vary throughout the simulation. This will skew the simulation tremendously. If the former line size check fails, signifying that the line size is

greater than 8*`num_of_s` the customer is turned away and the `overflow is` incremented.

If an event is a type departure, it signifies that the customer/event has already been serviced and that it is time to remove a customer that has been waiting in the `line` Queue to be serviced. This is done only if the line size is not zero meaning that there are still customers waiting to be served. The calculation for metrics is analogous to that done for an event that is type arrival, the only difference, however, is the calculation of wait time.
The `waitTime(current_time,s.getTime())` method is called and passed the parameters of current time and the arrival time of the event that was removed from the line. The current time in this context is the departure time of the previous event/customer which induces a removal from the line; so the wait time of a customer in line is calculated by subtracting the customer's arrival time from the current time. The instance of the customer is passed these metrics and the customer is then added to the ArrayList `waittime`. Moreover, if the line size is equal to zero, we increment the `num_of_s`.

This is essentially how the simulation functions and these steps are repeated until the PrioityQueue `shop` is empty.

Printing Metrics of CoffeeShop:
To calculate the total profit of the Coffee Shop, the `calculateTotalProfit(p,num_of_served)` method is called which is passed `p` (estimate profit per Customer) and the `num_of_served` . In the method, total profit is calculated by multiplying the two values together and setting that equal to type float variable `totalprofit` that is returned.

To calculate the total cost of operating the Coffee Shop for the day, the `calculateTotalCost(num_of_s,c)` method is called which is passed `num_of_s` and `c` . In the method, the total cost is computed by multiplying the two values an

setting it equal to a type float variable `totalcost`, which is returned by the method.

To calculate net profit of the Coffee Shop, the `calculateNetProfit(p,c,s,num_of_served)` method is invoked which is passed the parameters `p` , `c` , `s` , and `num_of_served` . In this method, the total cost of running the shop is subtracted from the total profit to get the net profit, which is assigned to a float type variable `netprofit` that is returned by the method.

To calculate the average wait time, the `avgWaitTime(waittime)` method is called which is passed the ArrayList of customers as a parameter. To calculate average wait time, an int type variable `total` is initiated to zero and a for loop is employed to loop through the ArrayList, acquiring each customer's wait time and adding that to the `total` . The average is then calculated by dividing `total` by the size of the ArrayList; This is set equal to the variable `avg_wait_time` which is returned by the method.

To calculate the maximum wait time, the `maxWaitTime(waittime)` method is called and passed the ArrayList of customers. To find the maximum time, a linear search (O(n)) is initiated with a for loop where a variable `max_wait_time` is consistently updated as the program loops through customer wait time and finds a value larger than the value stored in the variable `max_wait_time`.

To calculate the overflow rate, the `calculateOverFlow(num_of_served,overflow)` method is called where the number of customers served and those turned away are passed as parameters. To calculate overflow rate, the overflow is divided by the total number of customers (overflow + num_of_served) and further multiplied by 100. This value is set equal to the variable `overflow_rate` which is returned by the method.

Data and Analysis

Customer Arrival Time 1

Number of Cashiers	Net Profit (Dollars)	Avg Wait Time (s)
1	646	795
2	1074	575
3	1134	236
4	908	31
5	608	6

Table 1: Customer Arrival Time 1

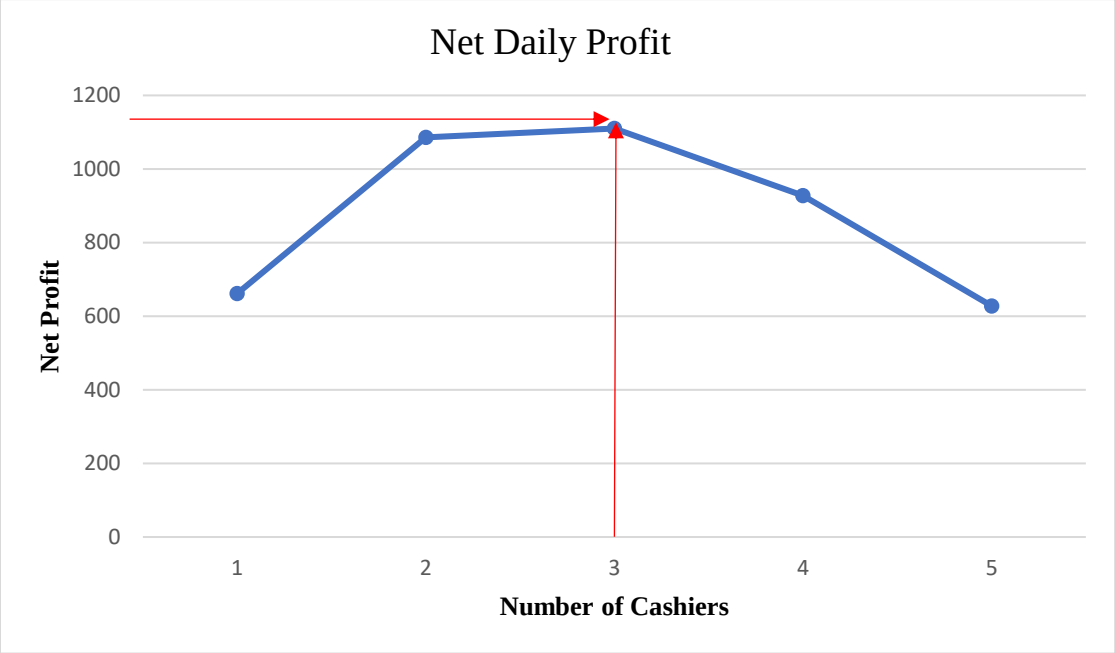

Figure 1: Number of Cashiers vs. Net Daily Profit – Customer Arrival Time 1

Customer Arrival Time 2

Number of Cashiers	Net Profit (Dollars)	Avg Wait Time (s)
1	662	835
2	1086	490
3	1110	313
4	928	94
5	628	12

Table 2: Customer Arrival Time 2

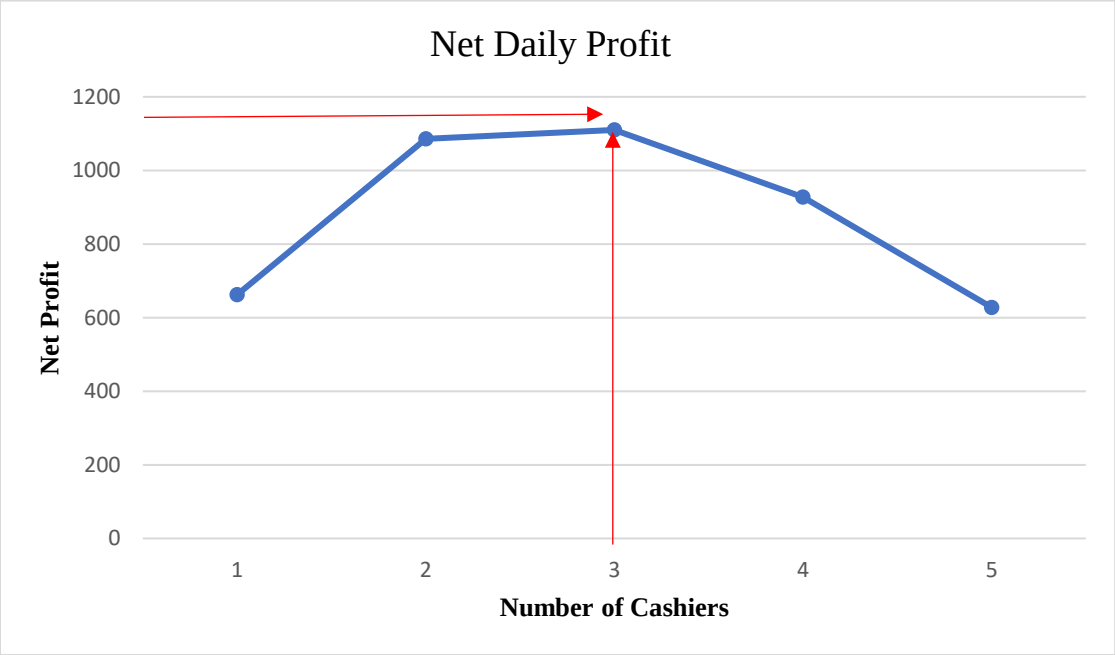

Figure 2: Number of Cashiers vs. Net Daily Profit – Customer Arrival Time 2

Analysis:

The simulation was run on two of the Customer Arrival input files given by the instructor. When analyzing both results generated by the simulation, similar patterns in wait time and net profit are evident.

As shown in Table 1 and Figure 1 of Customer Arrival Time 1, the net profit increases from $646 to $1134 as the number cashiers go from 1 to 3. The wait times also significantly decrease as it goes from 765 s to 236s. However, as the

number of cashiers increases from 3 to 4 and from 4 to 5, there is a sharp decrease in net profits as it goes from $1134 at 3 cashiers to $608 at 5 cashiers. The same declining pattern in profits is seen in Table 2 and Figure 2; the maximum net profit is also attained when there are three cashiers. These observations are supported by the concave down shape depicted both in Figure 1 and 2, where an inflection point exists at three because its where both graphs go from sharply increasing profits to decreasing profits.

Conclusion

Based on the results obtained from the simulation, the owners of the Coffee Shop should hire a maximum of three cashiers if they want to attain maximum profit. Although hiring more cashiers will yield lesser wait times and guarantee that all customers that enter the shop are served, the cost to maintain the additional cashiers will drive the profits of the shop down. This is because the profit per customer is constant as the cost of maintaining the shop and its cashier's increases.

This hence disproves the hypothesis I made earlier in my introduction; The wait times did significantly decrease as more cashiers were hired but the net profit also decreased for the reasons mentioned above.

References

1. API, J. (2018, June 23). Class PriorityQueue. Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html
2. API, J. (2018, June 23). Interface Queue. Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html
3. Weiss, Mark Allen. Data Structures & Problem-Solving Using Java. Pearson Education, 2010. Chapter 13: Simulation