Cameron Zurmuhl

Professor Xia

CS150

10 March 2017

Project 1 Report

I. Introduction

The project's goal is to simulate a 960-minute work day at a coffee shop on College Hill using discrete, event-driven simulation. The arrival and service times (if applicable) of the customers are calculated using a Poisson distribution. By specifying which events occur at calculated times, a PriorityQueue ("Java API", 2016) is used to process the events (entering/leaving) in the order they occur, due to its natural ordering structure (Weiss, 280). The idea of the simulation is to be as real-world as possible: a queue will place incoming customers in a line, and when a cashier is available, the customer will go to the cashier, get served, and leave. The process repeats for the next customer in line. If the line grows too large (8 * the number of cashiers), incoming customer(s) will not enter the store and will count as overflow. Each customer has a statistic—an arrival time, a waiting time in line, a service time, and a departure time. All customers are stored in appropriate lists for statistical calculations. For given values of the average customer arrival interval (lambda), I will simulate, on an eight-run average, the total net profit for number of cashiers 1-10, fixing the average service interval to .3 customers/minute, the cost per cashier at $300.00, and the profit per customer to $2.00. I will analyze the results for each given lambda to find the optimal cashier number for that lambda, which will correlate with the most profit. For this simulation, I assumed the Poisson distribution was the appropriate distribution to use. My hypothesis is that as the customer arrival rate increases, the optimal number of cashiers will also increase.

II. Approach

The approach I used was an event-driven simulation. This is opposite to looping through the minutes and analyzing what happens at each minute (Weiss, 509). I created a specific Event class, where the instances of the class will be stored in a PriorityQueue. An event class includes an integer specifying the customer the event applies to, an integer to specify what the event is (arrival or departure), and a time the event happens. The class must implement comparable for the PriorityQueue: the compareTo override is based on the event's time. To supplement the event class, I created a Customer class. The Customer class has an identifying integer, which corresponds to the customer integer in the event class, as well as other statistical based instance variables, mentioned in the introduction. As the customers come into the store, they are added to a Queue ("Java API", 2016), implemented with a LinkedList ("Java API", 2016). I implemented the queue in a CustomerLine class that adds customers to a line, polls from the line, and reports/changes the status of its Boolean variable that reports if it is full. If the variable is false, customers can be added to the queue.

The main simulation happens in class OperationSim. In OperationSim, all the arrival times of customers are calculated and stored in the PriorityQueue before the simulation loop. The first operation in the loop is polling from the head of the PriorityQueue. The event classification is asked--arrival or departure. If it is a departure, the available cashier number is increased, and that customer's departure time is set. If it is an arrival, the question is asked if the CustomerLine is full. If it is full, the customer is sent away as overdraft, and added to an overdraft ArrayList and a general ArrayList ("Java API", 2016). The customer's service time and wait time is set to zero. The departure time will be the arrival time. If the CustomerLine is not full, the customer is added to the CustomerLine, and to the ongoing, general Customer ArrayList. If there are customers in the CustomerLine with an available cashier, an upcoming customer is polled from the head of the CustomerLine. That customer goes to the cashier (decreasing available cashiers), and his wait time and service times are calculated. Last, this customer's leaving will be created as a new event with the appropriate time, and added to the PriorityQueue. After 960 minutes, only customers in the line before closing can be served (no more entries).

III. Methods

An ExperimentController class uses OperationSim's simulation method, but repeats the simulation for a specified number of runs. In the ExperimentController class, the method that runs multiple simulations takes all the project's assigned parameters (number of cashiers, service rate, arrival rate, cost per cashier, and profit per customer), and calls OperationSim's constructor appropriately, up to a specified amount of times by user input. I decided to run my simulations eight times, compounding the statistics for each run, and taking a proper average at the end. All trial statistics are printed out at the end of each trial to the terminal. The averaged statistics are printed out once after trial eight to the terminal. In addition to the printouts, I have Java's PrintWriter ("Java API", 2016) class report each customer's statistic, each trial's total statistics, and the final averaged statistics, to a .csv file. I chose eight runs to have a balance of run time and averaged quantity.

After running the experiment for the given parameters, I observed the profits per cashier number, per lambda, and created tables in a main spreadsheet.

IV. Data and Analysis

The following graphs were created after the averaged trials:

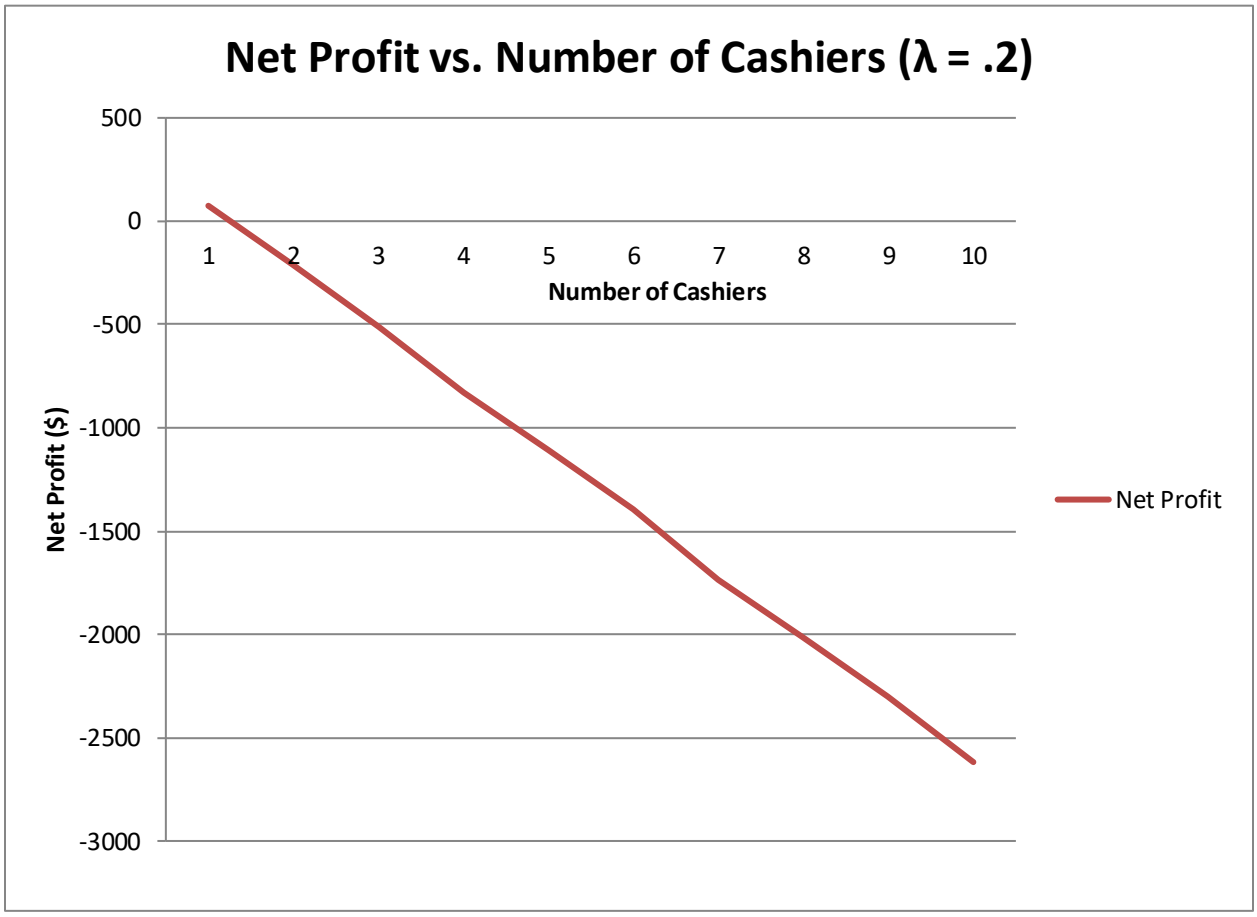

Chart 1: Net Profit vs Number of Cashiers when lambda is .2

The obtained results were expected. With so few customers arriving to the shop, higher number of cashiers are going to increase expenditures while profits are low, resulting in tremendous losses. The optimal number of cashiers in this situation is just one. The overflow rate with one cashier was low (1%), and the average waiting time was about 5 minutes. The net profit was $74.50.

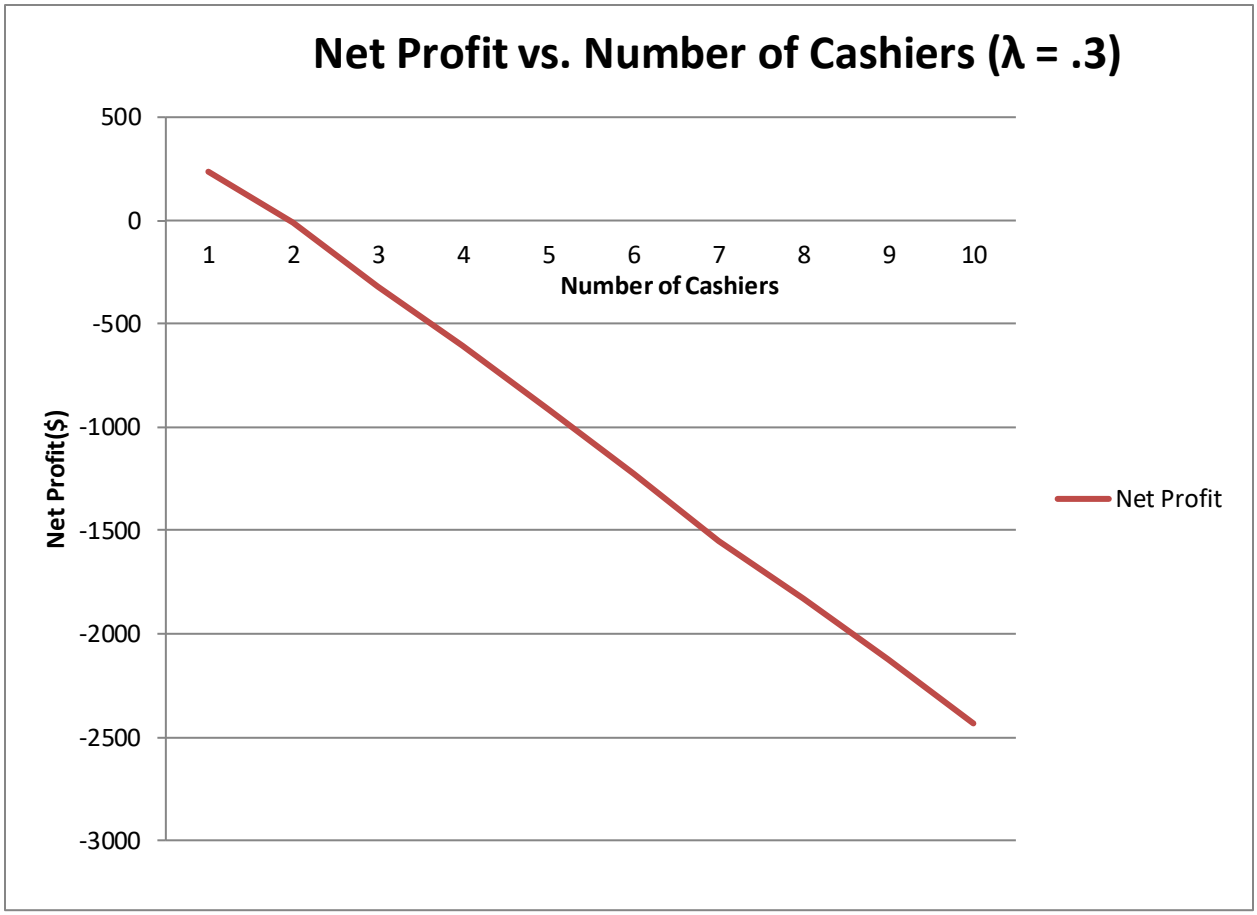

Chart 2: Net Profit vs Number of Cashiers when lambda is .3

These results are also plausible. The average customer arrival rate does not increase significantly from the last. Only one cashier is needed to handle so few customers. For one cashier, the averaged overflow rate increased to about 8%, the waiting time to 11.64 minutes, and the profit to $236.25 due to more customers.

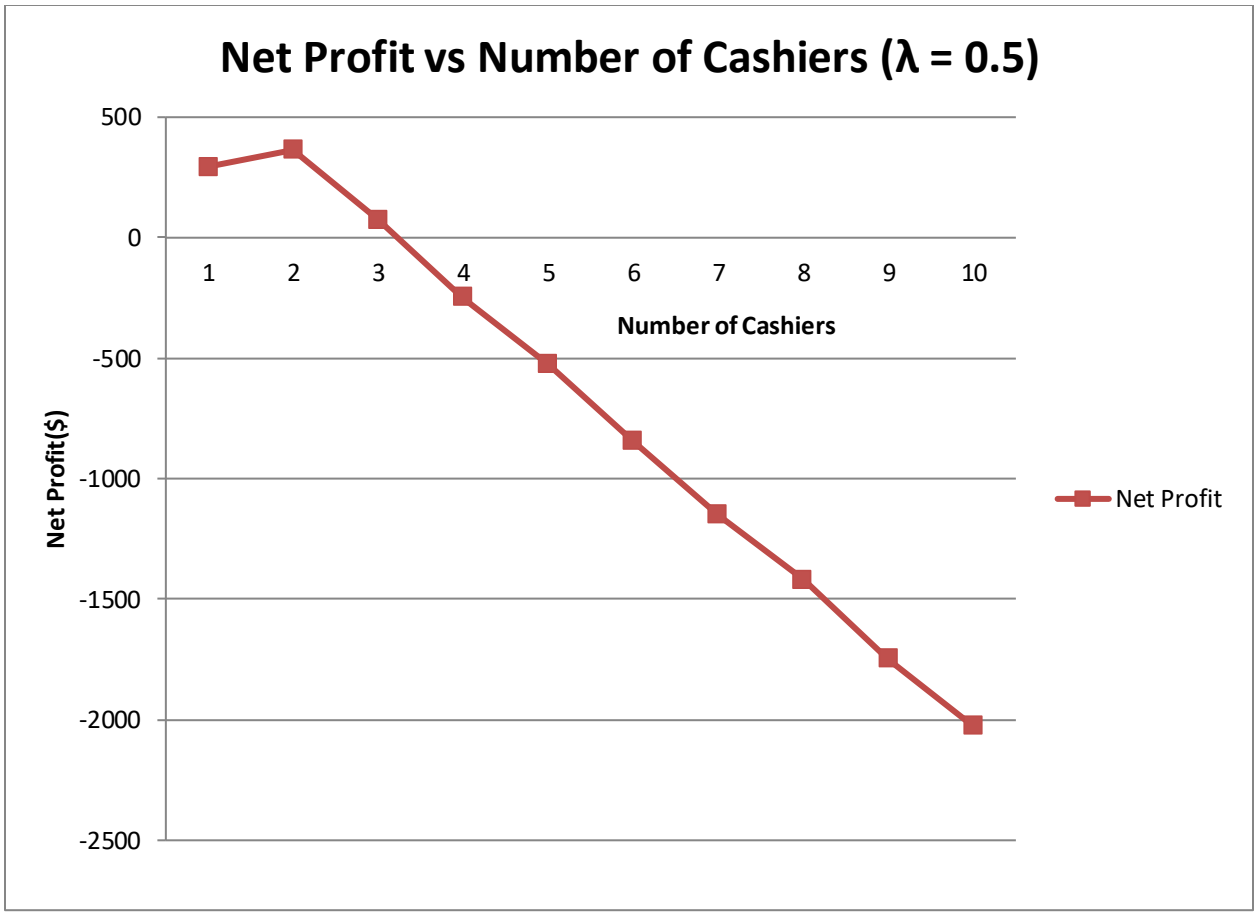

Chart 3: Net Profit vs Number of Cashiers when lambda is .5

The curve starts to look like a unimodal distribution. The profits increase as number of cashiers increase, reach a peak, and then decline. For this simulation, the peak occurs when there are two cashiers processing customers arriving once every other minute, on average. The optimal average waiting time is 6.17 minutes, the overflow rate is 1%, and the net profit is $363.75 for this configuration.

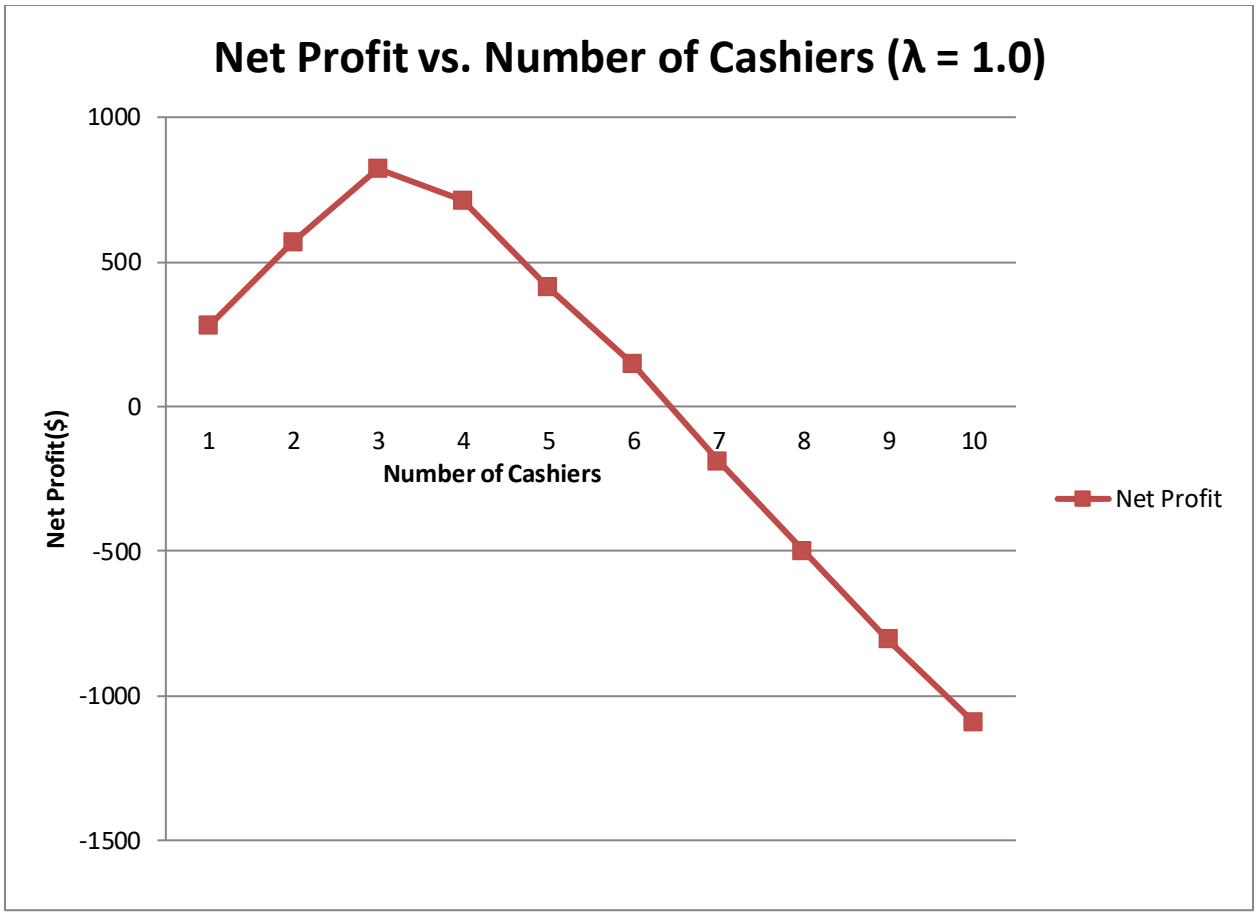

Chart 4: Net Profit vs Number of Cashiers when lambda is 1.0

Here the unimodal distribution becomes clearer. For one customer arriving per minute, on average, three cashiers are optimal. The averaged statistics are 9% overflow, 17.3-minute waiting time, and $821.50 net profit.

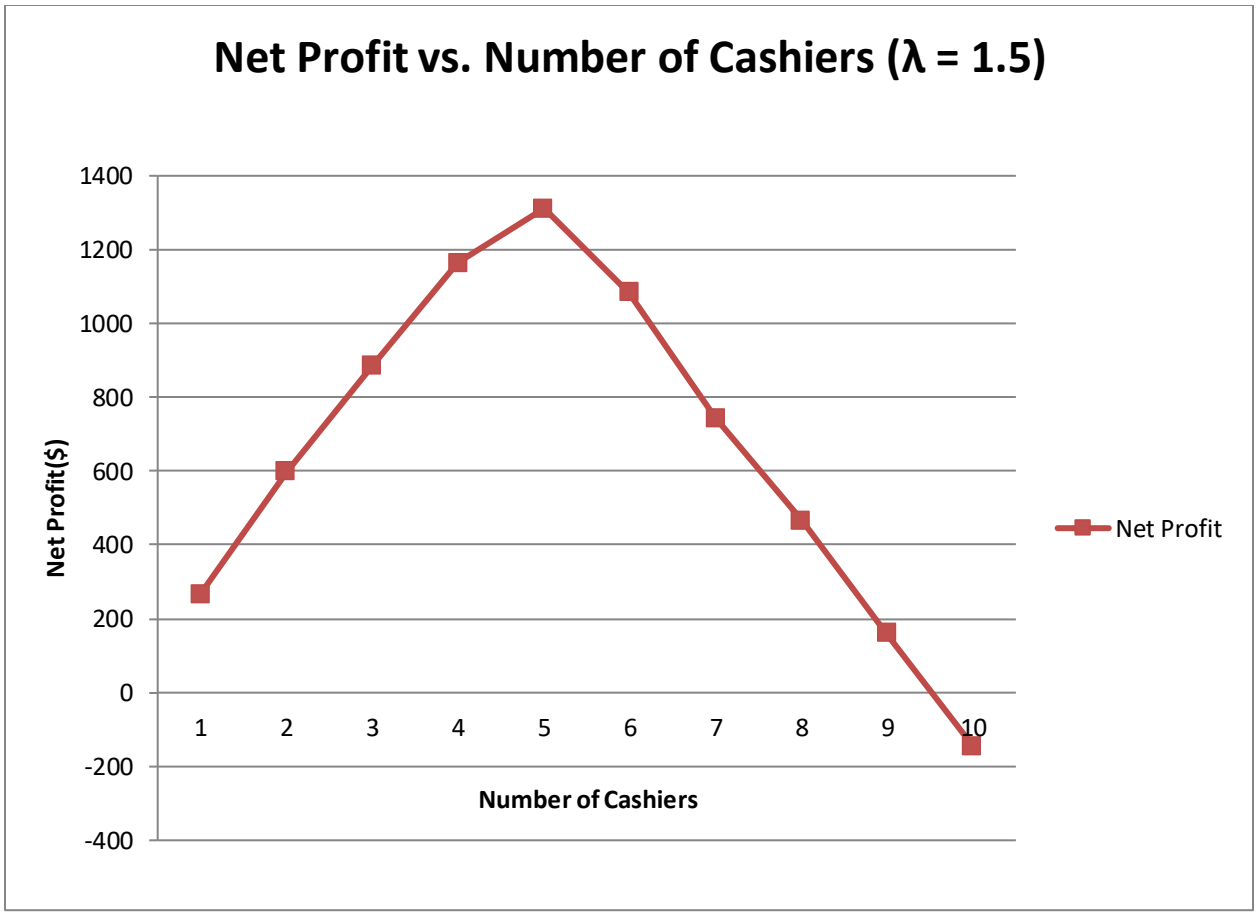

Chart 5: Net Profit vs Number of Cashiers when lambda is 1.5

Here the average customer arrival rate starts to pick up, requiring a higher number of cashiers. Profits increase as cashier numbers increase, reaching a stark peek at five cashiers. When the shop becomes too cashier heavy, the income can't recompense for their salaries. The optimal averaged statistics happen with five cashiers. The result are a 2% overflow, a waiting time of 12 minutes, and a net profit of $1311.75. Notice that as lambda increases, the optimal peeks shift to the right.

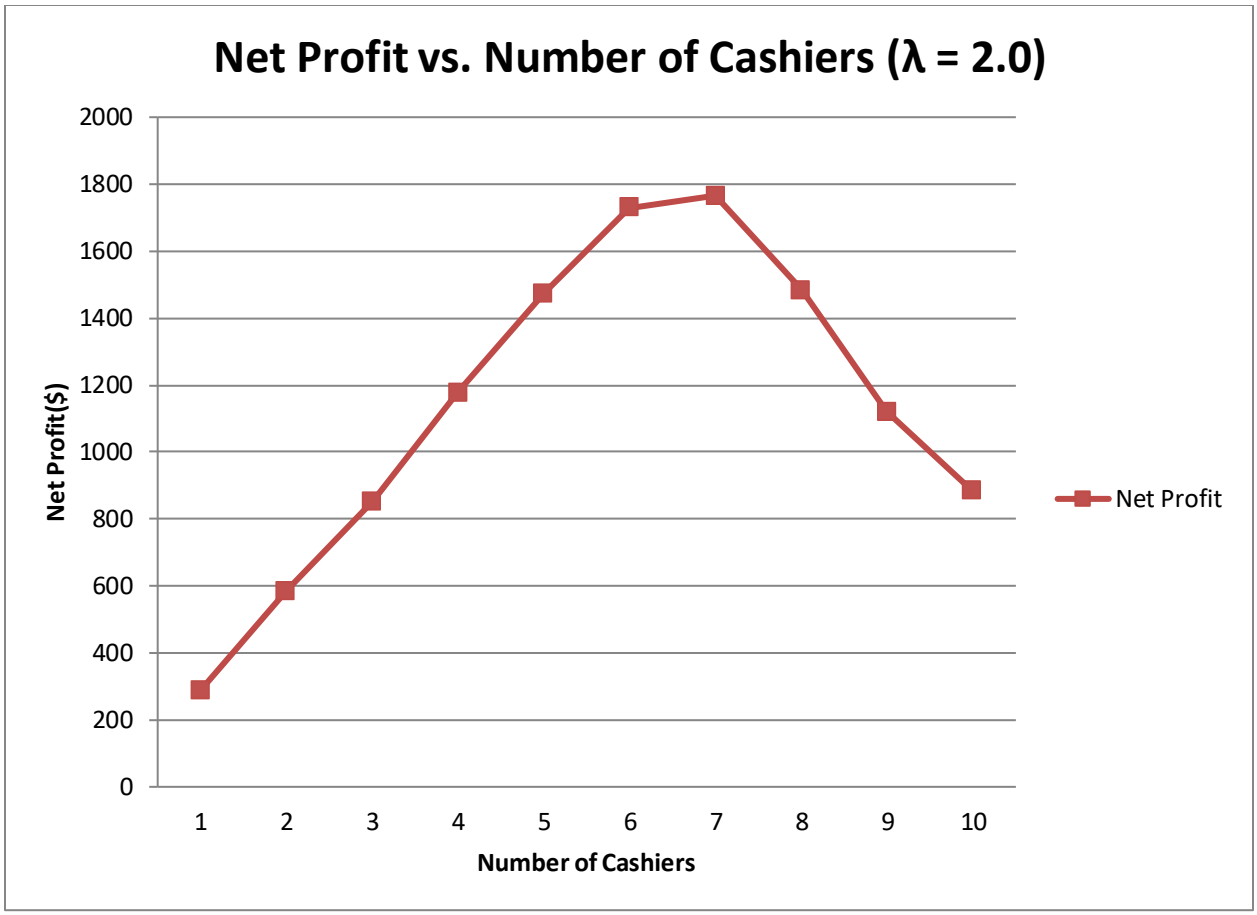

Chart 6: Net Profit vs Number of Cashiers when lambda is 2.0

When lambda is maximized to two customers per minute on average, more cashiers are needed to process them. If too few are available, the overflow rate explodes. Interesting enough, this is the only configuration where all cashier options result in positive profits because of the high volume of customers. In the optimal scenario with seven cashiers, the averaged statistics are a 0% overflow rate, 5.09-minute waiting time, and a net profit of $1765.75

The following graph is thus created to show the optimal number of cashiers for each lambda:

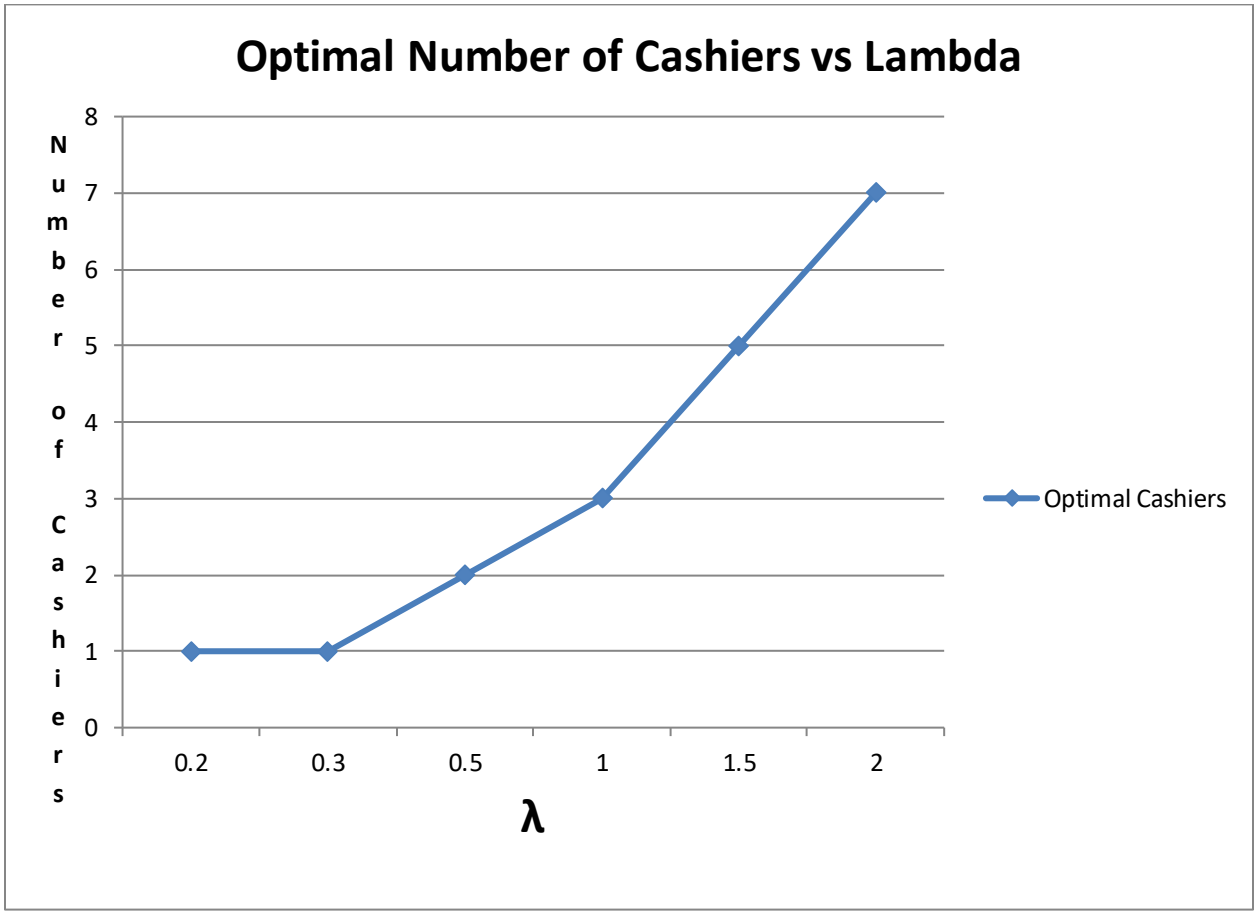

Chart 7: Optimal Number of Cashiers for each Lambda

V. Conclusion

For small increases in lambda (<0.1), there are only marginal differences in profit gains, and the optimal cashier number remains constant; thus, part of my hypothesis is disproved. However, when lambda grows by .2 or more, more cashiers are needed to handle the customer flow. The graphs of profits versus cashier numbers for each lambda begin to show a unimodal distribution as lambda increases, signaling a clear optimal situation. If cashier numbers are too unbalanced with the average customer arrival interval, profits will suffer. The optimal number of cashiers for lambda equals .2 is one, .3 is one, .5 is two, 1.0 is three, 1.5 is five, and 2.0 is seven.

VI. References

1. “Package java.util (Java Platform SE 8)” Oracle, 2016. Web. 8 March. 2017. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
2. “Package java.util (Java Platform SE 8)” Oracle, 2016. Web. 10 March. 2017. https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
3. “Package java.io (Java Platform SE 8) ” Oracle, 2016, Web. 10 March. 2017. https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html

4. “Package java.util (Java Platform SE 8)” Oracle, 2016. Web. 8 March. 2017. https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html
5. “Package java.util (Java Platform SE 8)” Oracle, 2016. Web. 8 March. 2017. https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

6. Weiss, Mark Allen. *Data Structures & Problem Solving Using Java*. 4th. Boston: Pearson Education, Inc., 2010. print.