Burrito Run
A Simulation Study on Chipotle Queues

Ryan Lattanzi

December 10, 2018

Abstract

In this simulation study, we compare two algorithms of queuing systems from Sheldon Ross's *Simulation*. We apply these algorithms to the popular restaurant Chipotle, where our question lies in finding the expected customer waiting time. The first system is how the restaurant currently operates: one queue with two servers in series. The second is again a single queue but with two parallel servers. After developing code for these algorithms in Julia, we found that the second system has a smaller expected waiting time. Our conclusion is that this decrease in expected waiting time is not worth altering the restaurants, but we leave it to corporate Chipotle to decide for themselves.

1 Introduction

Chipotle is a classic college-town go-to for those who want a quick, healthy, and cheap meal. With its continued growth in popularity, it is natural to investigate if there is any way to improve the burrito-creating efficiency so as to serve more customers, especially during peak hours. Note that although our aim is directed at Chipotle, this problem can be generalized to other fast-paced restaurants such as Panda Express, etc.

Our investigation can be completed via simulation, pulling algorithms from Sheldon Ross's *Simulation* (Fifth Edition)[1]. When we introduce our problem, we will also point out its assumptions and limitations. These can hinder the authenticity of our results if they are too strong. Then, we describe in detail the way each algorithm works. Finally, we present our results and conclusions.

2 Problem Description

In this simulation study, we ask the question: what is the expected waiting time of customers during lunchtime? That is, what is the average time that a customer will spend in line and get their food prepared? We take lunchtime to be the two hour period from 11:30 am to 1:30 pm. As mentioned, we will compare two algorithms from Sheldon Ross's *Simulation* that represents the Chipotle process as is, and a different process to see which yield lower expected waiting time. In particular, Chipotle currently runs according to a "queuing system with two servers in series" (Section 7.3 from [1]). Here we have one queue, and the customer proceeds from the first server in the burrito (or bowl) assembly line to the next to get their food. The comparative process accords to a "queuing system with two parallel servers" (Section 7.4 from [1]). In this case, there is still one queue, but there are two assembly lines with one server each for the customers to have their food prepared. Intuitively, we can infer that the second process will have a shorter waiting time. Our simulation will in fact confirm this, but also pose another question: Is the shorter time worth the money and effort of having two assembly lines?

2.1 Assumptions

Our base assumption is that customers arrive according to a homogeneous Poisson process with rate 70 customers per hour (7/6 customers per minute). We came up with this rate from collected data of another Chipotle project [2]. This team counted 68 customer arrivals of a particular Chipotle from 11:27 am to 12:39 pm. We assume this rate would stay approximately constant for another hour of lunchtime which is why we assume a homogeneous process rather than a nonhomogeneous one.

Another user-inputted value that we must assume is the serving time of each server, which we assume follow a normal distribution. These distributions differ in each model because of the different work loads for the servers. In the current Chipotle scenario, we claim each server takes on average 0.75 minutes to serve with standard deviation 0.25 minutes. That is, we assume a $\mathcal{N}(0.75, 0.25)$ distribution. In the second model, we assume a $\mathcal{N}(1.5, 0.5)$ distribution. These values were not chosen arbitrarily; they were derived by taking the average of the total time of the order from [2].

The last assumption is that after the customer's burrito is made, he will immediately depart the system. In real life, the customer would have to pay at a register, but we neglect this part since payment time is usually very short. Our simulation can easily be extended to include this factor (by pulling the algorithm in section 7.2 from [1]), but we do not see it necessary.

2.2 Limitations

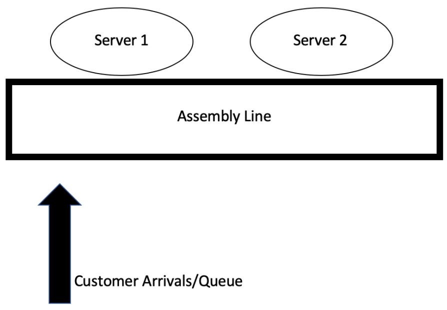
The first limitation is assuming a homogeneous Poisson process. It might be the case that the second hour of our lunch time does not have the same rate as the first. Thus, by the randomness of a nonhomogeneous process, the waiting times could be either longer or shorter. Another limitation lies in the data used: not all Chipotles will be as popular as the one in which the data was taken, and some may be even more popular. Hence, our arrival rate could change which would effect the waiting times.

The server time assumptions can also pose as a limitation. However, even with these altered, the essence of the problem still holds; comparing the two models will still be valid with the times growing or shrinking in proportion. Perhaps a more in depth limitation is assuming the average and standard deviation of server times in the second process are double that of the first process. These, if significantly skewed, could potentially lead to a misrepresentation of comparative waiting times.

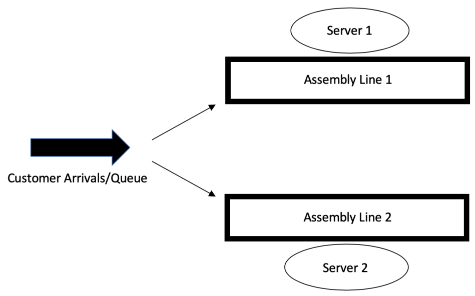
Although limitations rendered by our assumptions are present, we claim they are not strong enough to hinder our experiment. Our assumptions allow us to grasp the essence of the problem statement.

3 Algorithm Descriptions

Visuals of both algorithms are shown below in **Figure 1**:



(a) Two Servers in Series



(b) Two Parallel Servers

Figure 1: Illustrations of algorithms where (a) shows the current model, and (b) shows the comparative model.

3.1 Single Queue with Two Servers in Series

For this system, as the customer arrives, he will either enter into service with the server 1, or add to the queue. After completing the first service, he will move on to the second service; again either entering straight into the service, or waiting for server 2 to be free. Upon completion of both services, (i.e. the burrito or bowl is finished) the customer will leave the assembly line and (presumably) eat.

The variables for keeping track of the events are as follows:

- t - Time Variable
- $n1, n2$ - Steady State Variables

  - These keep track of the number of customers at or in the line of server 1 or 2, respectively.
- $Arrive, Dep$ - Output Variables
  - We initialize these as empty lists and append arrival and departure times of customers as they come and go.
- tA, tF, tD - Event List
  - tA - Time of the the next arrival
  - tF - Time that the customer is finished with server 1
  - tD - Time that the customer is finished with server 2 and leaves the system
  - Note: if there is no customer at server 1 or 2 at any given time, we set $tF = \infty$ or $tA = \infty$. The reason for this will become apparent when describing the algorithm.

Before we begin with the algorithm, it is necessary to clear up any uncertainty of the event list variables. These three variables are generated according to the assumed distributions (which we have established in Section 2.1). Thus, they are not the event times themselves, but rather the time until the event that the variable represents. For example, assume that customer k has arrived at time T_k and is being served by server 1. To find the time server 1 will complete his task, we must generate a random variable that adheres to the assumed server time distribution, $X \sim \mathcal{N}(0.75, 0.25)$ in this case, and set $tF = T_k + X$. We analogously generate the other two event variables. Now, to the algorithm.

Initially, we set $t = n1 = n2 = 0$, $Arrive$ and Dep as empty lists, and $tF = tD = \infty$ to ensure the first event will be a customer arrival (see Case 1 below). Recall that $t = 0$ represents our starting time, 11:30 am, and any time t after represents the number of minutes after 11:30 am. Now, it is time to generate our first arrival time according to the homogeneous Poisson process. The algorithm then follows a tedious process of updating all variables depending on the generated times of the event list variables. There are three cases:

1. The arrival of a customer is the next event (i.e. $tA = \min(tA, tF, tD)$).
2. A customer finishing with server 1 and moving to server 2 is the next event (i.e. $tF = \min(tA, tF, tD)$).
3. A customer's burrito is finished and he leaves the system (i.e. $tD = \min(tA, tF, tD)$).

We skip the details of each variable update since they are relatively straight forward, but invite the reader to review the code found in the appendix (Section 7.1), or consult [1]. What is worth mentioning is the method of ending each lunch period, which we will call a "day" from now on. We do not want to accept any customers that arrive beyond $t = 120$, since this corresponds to 1:30 pm. However, we will still continue to serve those who arrived before 1:30 pm. In order to accomplish this, if the next arrival time $tA > 120$, we set $tA = \infty$ so as to never to return to case 1. Then, we continue running the simulation until no customers are left (n1 = n2 = 0).

3.2 Single Queue with Two Parallel Servers

Customers arriving in this system will enter a single queue and proceed to one of the two assembly lines manned by one server each. Note that the customers may not arrive and depart in the same order. This algorithm more complicated to implement than that of section 3.1 since there are more variables to keep track of. They are listed below:

- t - Time Variable
- n, i_1, i_2 - System State Variables
  - n represents the total number of customers either in line or being served.
  - i_1 indicates the i^{th} customer is with server 1
  - i_2 indicates the i^{th} customer is with server 2
  - Note: i_1 and i_2 can be thought of as indicator variables which are temporary and updated as customers complete either service 1 or 2.
  - Note: While coding this algorithm, we will see that it is convenient to represent these states in a list, $SS = [n, i_1, i_2]$.
- N_A, C_1, C_2 - Counter Variables
  - N_A counts the number of customer arrivals, and is the key element for keeping our collection of departure times in order.
  - C_1 and C_2 keep track of the number of customers served by server 1 and server 2, respectively.

- *Arrivals*, *Departures* - Output variables (Same as Section 3.1)
- $tA, t1, t2$ - Event List (Same as Section 3.1)

To initialize this system, set $t = N_A = C_1 = C_2 = 0$, $SS = [0, 0, 0]$, and $t1 = t2 = \infty$ similar to the previous system. Then, we generate the first arrival time according to our homogeneous Poisson process. We again have three cases, but they differ from those in section 3.1:

1. The arrival of a customer is the next event (i.e. $tA = \min(tA, t1, t2)$).
2. The departure of a customer from server 1 (i.e. $t1 = \min(tA, t1, t2)$).
3. The departure of a customer from server 2 (i.e. $t2 = \min(tA, t1, t2)$).

We will first clear up any uncertainties of keeping keeping our i_1, i_2, N_A, and departure times updated and in the correct order. The values of i_1 and i_2 correspond to i^{th} customers so they can take on any value of N_A - which keeps an index of the customer number for the day. Hence, the subscripts 1 or 2 merely indicate if customer i is with server 1 or 2. When the i^{th} customer leaves the system, we collect the departure time in terms of of the indicator variables i_1 or i_2. Updating other variables is not as straightforward as in the previous system, so we take some time to look at them in detail.

Case 1: Suppose we generate tA for the arrival of customer k (the k^{th} customer of the day). First, we would like to reset $t = tA$, indicating we are at the time of a customer arrival. Note that $N_A = k$, so we also collect this arrival time and store it in *Arrivals*. Within this case, we have four more cases.

1. If $SS = [0, 0, 0]$ (nobody is in the system), then we will send customer k to the first server and make the update $SS = [1, k, 0]$. Note this means $i_1 = k$, indicating customer k is with server 1.
2. If $SS = [1, j, 0]$ (some other customer j is the only customer in the system and is at server 1), we will send customer k to server 2: $SS = [2, j, k]$.
3. If $SS = [1, 0, j]$ (some other customer j is the only customer in the system and is at server 2), we will send customer k to server 1: $SS = [2, k, j]$.
4. If $n \geq 2$, then both servers are busy and customer k will join the queue: $SS = [n + 1, j, l]$ for some other customers j, l.

Case 2: Suppose our next event is customer k finishing service with server 1. We note that in this case $i_1 = k$. First, we set our $t = t1$ and increase our $C1$ value by one. Then, we collect the departure time of customer k (in the code, we collect the departure time of i_1 but recall $i_1 = k$). We have three more cases now.

1. If $n = 1$, customer k was the only customer in the system. Since he is leaving now, we will have no more customers in the system. So, we reset $SS = [0, 0, 0]$ and $t1 = \infty$ to ensure our next event will be the arrival of a new customer.
2. If $n = 2$, then there is still going to be some customer j at server 2, so we update $SS = [1, 0, j]$ and $t1 = \infty$ to ensure our next event will either be a new customer arrival or customer j finishing with server 2.
3. If $n > 2$ then we have customers in the queue. Our next move must be to put the next person in line to server 1. To accomplish this, suppose we have $SS = [n, k, j]$, where j is some other customer. We let $m = \max\{i, j\}$, so that $m+1$ is the next customer in line, and update $SS = [n-1, m+1, j]$. Now, it is left to generate a new $t1$ that will be the time that customer $m + 1$ is done with server 1.

Case 3: This case in analogous to Case 2, except we change all $t1$ to $t2$ and hence neglect the details.

We terminate the day when $tA = \infty$ (we set $tA = \infty$, when $tA > 120$) and $SS = [0, 0, 0]$ as before.

4 Simulation and Results

4.1 Statistical Criterion

The simulation study consisted not only of coding the two algorithms, but also performing statistical analysis to ensure we have achieved a sufficiently accurate estimate of the average waiting time. This involved running each algorithm until the collected data achieved some criterion which will be explained shortly. To make the code efficient, we created functions of each algorithm and called them whenever necessary. Furthermore, based on the statistical analysis, we were able to extract more information from our simulation that could be useful. For example, we counted the number of runs it took to achieve our sufficient estimate, a confidence interval for our estimate, the average number of customers that visited per day, and the average time after 1:30 pm that the last customer was served. Additionally in the second system, we calculated the average number of customers each server helped per day.

To develop a statistical analysis criterion, we initially naively chose our 95% average waiting time confidence bound to be ± 5 minutes. However, after observing the results, we realized our bound was far too large and changed it to ± 5 seconds (or $\approx \pm 0.083$ minutes). This yielded more interpretable results. To achieve this confidence interval, the interval length would be $l = 0.166$. We would continue to run the simulation until

$$2z_{0.025}\frac{S}{\sqrt{k}} < 0.166$$

where k is the number of simulations, S is the standard deviation of the observed data after k runs, and $z_{0.025} = 1.96$. If this condition is satisfied after n runs, our final confidence interval would take the form

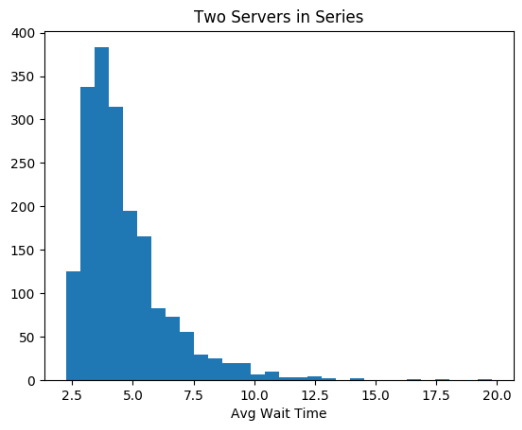
$$\left[\overline{x} - z_{0.025}\frac{s}{\sqrt{n}}, \overline{x} + z_{0.025}\frac{s}{\sqrt{n}}\right]$$

where $\overline{x}$ and s are the observed average waiting time and standard deviation after n iterations.

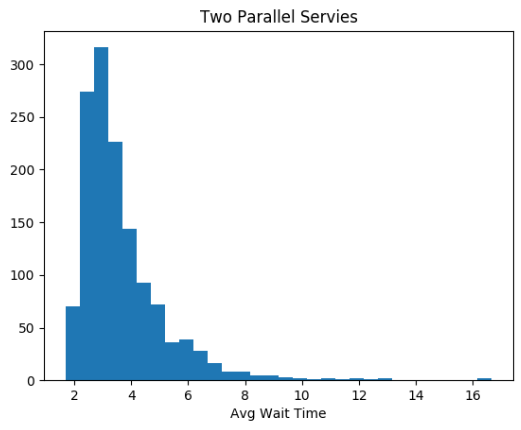
To execute this, we first ran the simulation 100 times to collect a sample starting data. Using this sample, we implemented a while loop that continued to run the simulations until our criterion was achieved.

4.2 Results

Histograms of average waiting times are shown:



(a) Two Servers in Series



(b) Two Parallel Servers

Figure 2: Average waiting time histograms of the two systems.

We can see the systems follow similar distributions, but it looks as though the mean for the first system is slightly higher than the second. We also show the output of our code to confirm:

```
We would like our avg waiting time to be within plus/minus 0.083 mins (plus/minus 5 seconds)
Hence, our confidence interval will have length:  0.166
SIMULATION 1:
This simulation ran 1886 times.
Expected waiting time:  4.621981876408666
95 % confidence interval for expected waiting time:  [4.53901, 4.70495]
Average number of customers after 1886 runs:  139.9305408271474
Average number of minutes after the two hour period that the last customer leaves:4.4964303437097675
SIMULATION 2:
This simulation ran 1080 times.
```

```
Expected waiting time:  3.6132661489027047
95 % confidence interval for expected waiting time:  [3.53028, 3.69626]
Average number of customers after 1080 runs:  140.61481481481482
Average number of minutes after the two hour period that the last customer leaves:  3.37004015769226
Server 1 served on average 72.46574074074074customers.
Server 2 served on average 68.14907407407408customers.
```

Hence, the average waiting time for the current Chipotle system is about 4.6 minutes. This estimate was achieved after running 1886 simulated days. As we can see, the average waiting time for the second system is indeed shorter, at about 3.6 minutes. This system ran 1080 times. While we must keep in mind that these values will slightly change every time the code is run (due to the generation of random variables), these numbers are quite consistent because of our 95% confidence bound.

The average number of customers is seen to be just around 140 for both systems, which is expected because of our Poisson rate (70 customers/hour). We can also see the second system reduces the time after 1:30 pm that the last customer is served. Finally, we see that on average, server 1 will serve two more customers than server 2. Our hypothesis is that this results from the first case within Case 1, in which we send the customer to server 1 if the system is empty.

5 Conclusion and Further Questions

Results showed the average waiting time for the second system with two parallel servers is shorter than the current system being implemented by Chipotle. We are very confident that this is the case except for outliers. This result is expected because in the second system, customers do not have to wait for a slow customer - they can simply be served in the other assembly line. However, this result brings about the question, is the price of changing or adding a new assembly line worth the one minute decrease? Our answer is no, although we leave it to the financial experts to make the final call.

Further investigations of this project can be to modify the various assumptions made and see if these result in a larger gap between average waiting times of the systems. We can also add components to they system that were neglected, such as Chipotle's call-in order system. Here, customers call ahead of time to have their food prepared so it is ready for pickup when they arrive. These call-in orders are given priority to those who are in the queue, so it would be interesting to see how this effects the waiting time gap between the systems.

Overall, we learned a lot from this project, particularly figuring out logical steps to code in order to satisfy the algorithm provided. This included keeping track of variables, being creative with termination steps, and of course debugging.

6 Sources

[1] Ross, Sheldon M. Simulation. 5th ed., Elsevier, 2013.

[2] `https://sites.google.com/site/chipotlesim/data-collection`

7 Appendix

7.1 Function for Two Servers in Series

```
function sim1()
    t = 0; n1 = 0; n2 = 0; Na = 0; Nd = 0;
    tA = t - ((6/7)*log(rand()));
    tF = Inf; tD = Inf;
    Arriv = [];
    Dep = [];
    while true
        CASE 1
        if (tA == minimum([tA,tF,tD]))
            t = tA
            Na = Na + 1
            n1 = n1 + 1
            tA = t - ((6/7)*log(rand()))
            if tA > 120
                tA = Inf
            end
            if n1 == 1
                tF = t + abs(rand(Normal(0.75,0.25)))
            end
            push!(Arriv, t)
        CASE 2
        elseif (tF == minimum([tA,tF,tD]))
            t = tF
            n1 = n1-1
            n2 = n2+1
            if (n1 == 0)
                tF = Inf
            else
                u = rand()
                tF = t + abs(rand(Normal(0.75,0.25)))
            end
            if (n2 == 1)
                u = rand()
                tD = t + abs(rand(Normal(0.75,0.25)))
            end
        CASE 3
        elseif (tD == minimum([tA,tF,tD]))
            t = tD
            Nd = Nd + 1
            n2 = n2 - 1
            if n2 == 0
                tD = Inf
            elseif n2 > 0
                u = rand()
                tD = t + abs(rand(Normal(0.75,0.25)))
            end
            push!(Dep, t)
            if (tA > 120) && (n2 == 0) && (n1 == 0)
                break
            end
        end
    end
    return Arriv, Dep
end
```

7.2 Function for Two Parallel Servers

```
function sim2()
    t = 0;Na = 0;C1 = 0;C2 = 0;
    i1 = 0;i2 = 0;SS = [0,0,0]
    Arrivals = zeros(500);
    Departures = zeros(500);
    tA = t - ((6/7)*log(rand()));
    t1 = Inf;
    t2 = Inf;
    while true
        if (tA == minimum([tA,t1,t2]))
            t = tA
            Na = Na + 1
            Arrivals[Na] = t
            u = rand()
            tA = t - ((6/7)*log(u))
            if tA > 120
                tA = Inf
            end
            if (SS == [0,0,0])
                SS[1] = 1
                i1 = Na
                SS[2] = i1
                t1 = t + abs(rand(Normal(1.5,0.5)))
            elseif (SS[1] == 1) & (SS[2] != 0) & (SS[3] == 0)
                SS[1] = 2
                i2 = Na
                SS[3] = i2
                t2 = t + abs(rand(Normal(1.5,0.5)))
            elseif (SS[1] == 1) & (SS[2] == 0) & (SS[3] != 0)
                SS[1] = 2
                i1 = Na
                SS[2] = i1
                t1 = t + abs(rand(Normal(1.5,0.5)))
            elseif (SS[1] > 1)
                SS[1] = SS[1] + 1
            end
        elseif (t1 == minimum([tA,t1,t2]))
            t = t1
            C1 = C1 + 1
            Departures[i1] = t
            if (SS[1] == 1)
                SS = [0,0,0]
                t1 = Inf
            elseif (SS[1] == 2)
                SS[1] = 1
                SS[2] = 0
                t1 = Inf
            elseif (SS[1] > 2)
                m = maximum([i1,i2])
                SS[1] = SS[1] - 1
                i1 = m + 1
                SS[2] = i1
                t1 = t + abs(rand(Normal(1.5,0.5)))
            end
        elseif (t2 == minimum([tA,t1,t2]))
            t = t2
            C2 = C2 + 1
            Departures[i2] = t
            if (SS[1] == 1)
                SS = [0,0,0]
                t2 = Inf
            elseif (SS[1] == 2)
                SS[1] = 1
                SS[3] = 0
                t2 = Inf
            elseif (SS[1] > 2)
                m = maximum([i1, i2])
                SS[1] = SS[1] - 1
                i2 = m + 1
                SS[3] = i2
                t2 = t + abs(rand(Normal(1.5,0.5)))
            end
        end
        if (SS == [0,0,0]) & (tA == Inf)
            break
        end
    end
    filter!(e->e?0.0,Arrivals);
    filter!(e->e?0.0,Departures);
    return C1, C2, Arrivals, Departures
end
```