CX 4230 Mini Project 3 Report

Junhaeng Lee (jlee3571@gatech.edu)
Juntae Park (jpark3068@gatech.edu)

Introduction

This report is for CX 4230 Mini-project 3: NYC Bike Share. In this report, we'll explore discrete event simulation for New York City's bike-sharing service. We were given massive raw data about the probability of where bikes get picked up and returned at all stations, and essentially want to find the number of bikes needed for each station to meet full demand.

Task 1: Create a simulator

There are a couple of restrictions for this simulator, such as a time period that spans 24 hours, the number of station numbers, the number of riders, and an assumption that interarrival times constitute an autonomous, stationary, and independent stochastic process … etc. In order to construct such a model, we'll use create two classes that are essential to the simulation - Event class and FutureEventList. Objects of these classes are created whenever a new event happens, and this is what distinguishes discrete event simulation from continuous simulation. In the Event class, it contains information such as t (time), i (departing station), j (arriving station), f (event handler). The FutureEventList contains a list of Event objects that are scheduled for the future. This will handle the event based on ascending time manner. There's another important concept which is State. Although it's not a class, the State contains lots of crucial information for each station's current state. For example, it saves the information of num_bikes (a dictionary that has the station as the key and the number of available bikes at the station as a value), infoWaiter (a dictionary that has the station as the key and the queue of a waiter at the station as value), and waitTime (an integer that we keep track of in order to calculate average wait time). These components are very useful later on and will help us return a successful rental rate as well as an average wait time.

Now before we go over how we schedule events and handle those, we'll briefly go over how we're reading given data in CSV files. In the beginning, we read two separate CSV files that were given, and create a dictionary with start stations and their probabilities from one of them. Then, we calculate the probability of each station given a start station and the number of trips taken in "trips_stats.csv" file. It stores the resulting dictionary with start stations and their corresponding end stations and probabilities. We're using this approach as it'll be convenient to randomly generate starting points and ending points. This can be simply done in our randomGenerator function, by using random function from the Python library. We then proceed to create some helper functions such as static_vars, set_time, and schedule, which were provided in Demo.

We then wrote the simulation function, which is the main function that runs the simulation. It takes state and event_list as a parameter and returns a list that contains information such as successful rental rate as well as average waiting time. In order to retrieve this information, we need to iterate through every single event in event_list until we hit 24 hours time limit. For each event, we set the event time, call the event handler, and check the type of event. There are three types of events - which are arrived_startpoint, start_riding, and arrived_endpoint. For arrived_startpoint, we simply append each person who arrived at the start point into infoWaiter at the station. We check if we have any bikes at the station, and if there are, we schedule an event start_riding. In the start_riding event, we make sure there is a bike at the station as well as there is a person waiting for the bike at the station. Once we know that there's a person at the station as well as available bikes, we decrement num_bikes at the station and calculate the wait time (which is the difference between when they arrived at the station and actually start riding a bike. If someone just arrived and there's a bike available, then the wait time will be 0. We'll use this information later to calculate the average wait time). To determine the ride time, as stated in the PDF, we used a log-normally distributed value with mean = 2.78 and standard deviation = 0.619, which corresponds to around 16 minutes on average (let's call this ridetime). Once we've done that, we schedule another event arrived_endpoint at time = now + ridetime. There's one more thing to check in the start_riding event, which is if there are any more people that are waiting while there are available bikes. If this is the case, we schedule another event now and call start_riding again. The idea behind this is we want to make sure no one is waiting if they don't have to. Finally, in the arrived_endpoint event, we increment the num_bikes at the destination station. Since the rider is returning a bike to the destination station, we now have one more bike, and that means we can schedule another bide bike if there's any person waiting at the station. Therefore, we check if there's any person in infoWaiter, and if such a person exists, we schedule start_riding for that person at time = now. As stated in the PDF, for creating users, we made sure to follow exponentially distributed with λ = 2.38 riders per minute.

Task 2: A baseline experiment

For the baseline experiment, we ran the simulation with 3500 people in a 24-hour span. In order to get a 90% confidence interval estimate for the successful rental probability and average wait time, we essentially ran the simulation 30 times and recorded the successful rental probability and average wait time each time. We figured 30 will be a sufficient sample size as 30 is often used in different statistical model sampling. Once we had the list of 30 different values for both categories, we simply computed a 90% confidence interval estimate by calculating the mean and standard deviation. On one of our trials, we got (87.91, 89.55) for successful rental rates and (8.88, 11.47) for average wait time. In other words, we're 90% confident that the overall successful rental rate will be between 87.91% and 89.55%, and the average wait time is between 8.88 minutes and 11.47 minutes. The way we measured successful rental rate is quite simple - we keep track of the successful number of events of ridingcount and arrivedcount. Both ridingcount and arrivedcount are initialized as 0 in simulate function and get incremented every time upon successful corresponding event occurrence. The idea is rather intuitive; we want to see out of all the people who arrived at stations, how many people were able to ride a bike. The way we measured average wait time is by dividing total wait time by ridingcount. Here, it’s a similar idea where we initialize total wiat time as 0 and increment it in the event of start_riding as we explained above.

Task 3: An “idealized” experiment

Initially, we assumed that each station had one bike as a starting point. we devised a method to increase the starting bike count of stations that have riders waiting in the queue by one after one trial of simulation. So this idealized simulation runs until there are no more people waiting, i.e. everyone can successfully rent. However, because the starting and ending points are randomly assigned for each simulation, even if all riders successfully complete their rentals, there is a possibility of failure in the next simulation. Therefore, we set a condition that if the random simulations are successful for nine consecutive times, the idealized simulator stops and returns the minimum number of bikes required for each station. This means a success rate of over 90 percent.

```
{'South Waterfront Walkway - Sinatra Dr & 1 St': 36, 'Grove St PATH': 32, 'Hoboken
Terminal - Hudson St & Hudson Pl': 37, 'Hoboken Terminal - River St & Hudson Pl': 37,
'Newport Pkwy': 30, 'City Hall - Washington St & 1 St': 31, 'Newport PATH': 30, '12 St &
Sinatra Dr N': 26, 'Hoboken Ave at Monmouth St': 30, 'Marin Light Rail': 27, 'Hamilton
Park': 32, '14 St Ferry - 14 St & Shipyard Ln': 26, 'Liberty Light Rail': 25, 'Columbus Dr at
Exchange Pl': 26, 'Harborside': 23, '11 St & Washington St': 27, 'Washington St': 24, 'Sip
Ave': 26, 'Hudson St & 4 St': 24, '8 St & Washington St': 24, 'Madison St & 1 St': 24, 'City
Hall': 24, 'Warren St': 25, 'Newark Ave': 22, 'Columbus Park - Clinton St & 9 St': 22,
'Grand St & 14 St': 23, 'Church Sq Park - 5 St & Park Ave': 23, 'Columbus Drive': 21,
'Van Vorst Park': 20, 'Clinton St & Newark St': 24, 'Grand St': 19, 'Paulus Hook': 23,
'Manila & 1st': 21, '9 St HBLR - Jackson St & 8 St': 21, 'Bloomfield St & 15 St': 20, '4 St
& Grand St': 22, '7 St & Monroe St': 21, 'JC Medical Center': 20, 'Clinton St & 7 St': 20,
'Willow Ave & 12 St': 20, 'Morris Canal': 19, 'McGinley Square': 24, 'Brunswick & 6th':
20, 'Jersey & 3rd': 19, 'Brunswick St': 19, 'Baldwin at Montgomery': 20, 'Adams St & 2
St': 19, 'Southwest Park - Jackson St & Observer Hwy': 18, 'Marshall St & 2 St': 19,
'Journal Square': 20, 'Madison St & 10 St': 18, '6 St & Grand St': 20, 'Dixon Mills': 15,
'Lafayette Park': 16, 'Riverview Park': 19, 'Stevens - River Ter & 6 St': 18, 'Mama
Johnson Field - 4 St & Jackson St': 16, 'Pershing Field': 18, 'Hilltop': 20, 'Jersey & 6th
St': 16, 'Essex Light Rail': 16, 'Monmouth and 6th': 15, 'Oakland Ave': 21, 'Adams St &
11 St': 16, 'Bergen Ave': 17, 'Fairmount Ave': 18, 'Montgomery St': 15, 'Christ Hospital':
18, 'Astor Place': 17, 'Heights Elevator': 17, 'Lincoln Park': 14, 'Leonard Gordon Park':
19, 'Communipaw & Berry Lane': 13, '5 Corners Library': 13, 'Glenwood Ave': 14, 'Union
St': 12, 'Dey St': 11, 'Jackson Square': 10, 'Bergen Ave & Stegman St': 7, 'Grant Ave &
MLK Dr': 6, 'JCBS Depot': 2}
```

Above is an example of the result of one of our simulations; for each station, it indicates the number of required bikes in order to fully meet demand. As we can see, the range varies from 2 to 37 bikes per station. Note that our code terminates once we find over 90% success rate in 10 iterations, which means once we find 9 consecutive results of 100% accuracy we end the simulation. If we increase the success rate to an even higher percentage, such as 99%, we can expect even more “idea” scenarios and might slightly modify our final result.