NTNU
Kunnskap for en bedre verden

Department of Information Security and Communication Technology

TTM4110 - Dependability and Performance with Discrete Event Simulation

Fall 2024

Lab II
City Bus Simulations

CandID: 10031

Table of Contents

Task II.A: Implementation and Simulation Runs

II.A.1 Passenger Distribution

As mentioned in the task, the time T between the arrival of a passenger is modelled by a negative exponential distribution, hereby referred to as n.e.d. The probability density function of a n.e.d. is given by

$$f(t) = \lambda e^{-\lambda t}, t \geq 0$$

where

- λ is the intensity
- t is the time between passenger arrivals

The expected time between passengers, $E[T]$, is 2 minutes. Since the expected value is the inverse of the intensity, we can calculate λ.

$$E[T] = \frac{1}{\lambda} = 2 \leftrightarrow \lambda = \frac{1}{E[T]} = \frac{1}{2}$$

In Figure 1, you can see the behaviour of the PDF with $\lambda = \frac{1}{2}$. Notice the "long tail" which indicates that there always is a chance for a longer inter-arrival time than what is expected.

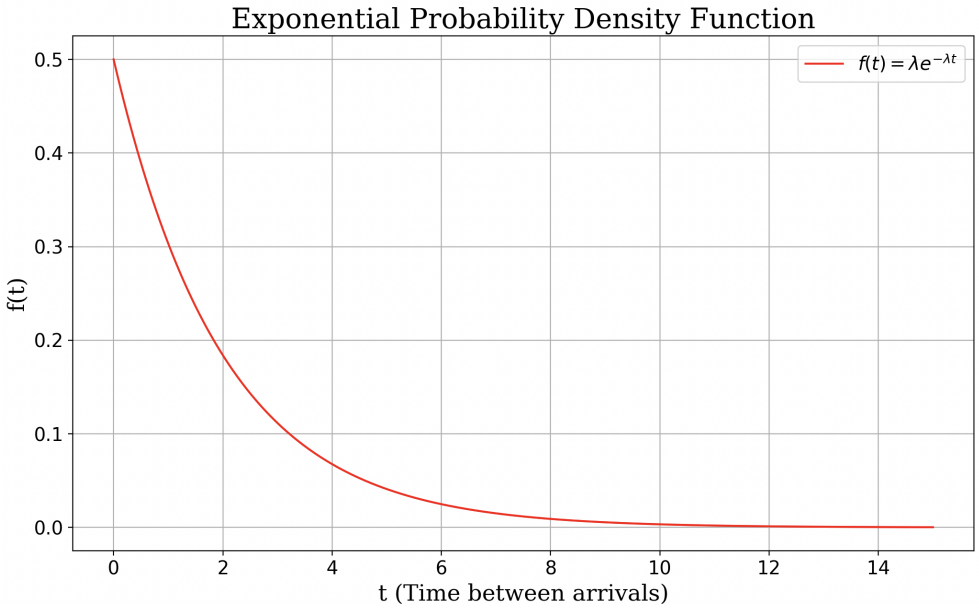


Figure 1: Probability Density function

In total, 2000 random samples was generated using two different methods:

- X_i: Implemented a manual method, using the inverse transformation technique, where a

uniform random variable U is transformed using $X_i = -\frac{1}{\lambda}\log(1-U)$ to follow the exponential distribution.

- Y_i: Used a `np.random.exponential`, a built-in function in Python.

Both sets were then sorted in increasing order and plotted, as visualised in Figure 2. See Appendix A to see how the distribution was generated.

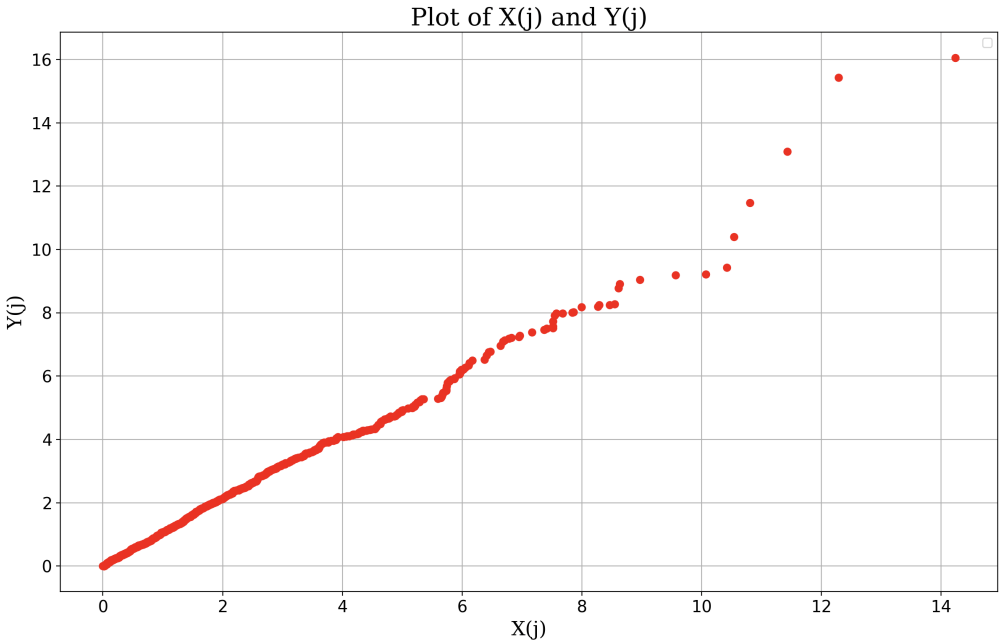


Figure 2: Distribution of randomly generated samples.

We can see that the plot shows a linear relationship between the X_i and Y_i values. This could indicate that both variables comes from the same distribution. We have a few samples at the higher values, but the samples are mainly centred around the expected value. This is consistent with the behaviour of the exponential distribution and its "long tail" effect, which gives the few cases of longer inter-arrival times. Even though shorter inter-arrival times are more common, the possibility of longer times still remain.

II.A.2 Representation of Routes

The routes are represented with a dictionary structure, where each key corresponds to a specific route. The associated value is a new dictionary with two keys, "S" and "R". "S" is an ordered list that represents the stops a bus visits along the given route. "R" is also a list, and represents the roads that connects the stops. The direction of the bus is indicated in the initial key, and the same route has a different order of the segments depending on the direction.

Listing 1 shows how the routes are represented. The eastbound bus on route 1 will traverse five

stops — endstops E_1 and E_3, and intermediate bus stops $S_{1,e}$, $S_{4,e}$, and $S_{6,e}$. The whole dictionary is available in Appendix B

```
routes = {
    "1 eastbound": {
        "S": ["E 1", "S 1 e", "S 4 e", "S 6 e", "E 3"],
        "R": ["R 1", "R 5", "R 8", "R 13"]
    }
}
```

Listing 1: Dictionary for route 1 going east

II.A.3 Implementing the Simplified Model

The simplified model consists of a passenger generator and a bus entity. In this model the passengers are represented as passive objects, meaning they have no direct influence on or interaction with the bus system. A passenger makes no decisions and has no purpose. It is generated by the system only to wait to be picked up. The passenger boards on allowance from the bus, and leaves when it is told.

It is not possible to isolate one specific passenger as they have no ID, and therefore we have no knowledge of how long a passenger have been waiting. This makes it difficult to make adjustments to reduce passenger travel time and increase their Quality of Experience.

As a result of the assumption that passengers are passive objects, the model's accuracy when reflecting the real world is somewhat limited. In reality, passengers could make an impact on the system by deciding to board or not based on capacity and timeliness, and entering and leaving at request. Because of this the model is missing important aspects that can influence the performance of the bus system and the passenger satisfaction.

The bus entity follows a predetermined route, stopping at defined bus stops along the way. When the bus arrives at a stop, it checks if there are any passengers waiting. If there are available seats, passengers are allowed to board. The bus continues this process until it reaches the end of its route, at which point passengers are dropped off. Since passengers have no specific IDs, there is no distinction between them based on where they board or where they intend to go.

Considering the simplicity of the model, I assume the implementation of `select route` is rather basic. This makes it hard to make adjustments with the purpose of enhancing utilisation. In my implementation, I am using random route selection. In task II.B.2 a more advanced route selection mechanism is implemented with the goal of enhancing bus utilisation and passenger travel time.

The implementation of the simplified model is available in Appendix C

II.A.4 Bus Utilisation

After running the simulation with different number of buses, we have an estimate for the utilization and standard error in the different cases. Listing 2 shows the output from simulating with a varying number of buses, and we can conclude that the utilisation decreases as number of buses increase. This happens because we have the same amount of passengers, and the load on each bus is reduced. Reasonably, we can assume that overall passenger waiting time is reduced as the number of buses increases, even though we have no measurement of it.

```
For 5 buses:
Average utilization across all simulations: 0.284
Standard error: 0.006
For 7 buses:
Average utilization across all simulations: 0.217
Standard error: 0.004
For 10 buses:
Average utilization across all simulations: 0.165
Standard error: 0.003
For 15 buses:
Average utilization across all simulations: 0.115
Standard error: 0.001
```

Listing 2: Output from simple model and $n_b = [5, 7, 10, 15]$

Figure 3 illustrates the average utilisation with error bars representing the standard error. The error bars are rather small, which indicates that the utilisation is consistent across the simulations.

Figure 3: Plot of Average Utilisation and Standard Error from the Simple Model

To calculate the average utilisation I keep track of the utilisation for each bus after leaving every stop, and then calculate the average utilisation for all those buses combined in each simulation. Lastly I combine the average utilisation for all of the simulations. This is the value you can see in Listing 2. I use the definition of utilisation that was defined in Lab I:

$$\text{Bus Utilisation} = \frac{\text{Number of passengers}}{\text{Number of seats on buss}}$$

I have chosen not to take the different travel times of the roads into consideration when calculating the bus utilisation. I did this because I assume the routes are static, meaning the buses follow the same set of routes throughout the simulation - the only thing we might change is the number of buses. So, because the bus cannot simply skip a stop or road because it has few passengers on a long road, it will not impact the utilisation.

Little's formula explains how the average arrival intensity, the average waiting time, and the average number of customers, relates to each other in a stationary system [Emstad et al. 2018, p. 104-105]. The formula is given by

$$\overline{N} = \overline{\lambda} * \hat{W}$$

where

- $\overline{N}$ is the average number of customers
- $\overline{\lambda}$ is the average arrival intensity
- $\hat{W}$ is the average time time

If we rearrange this, we can estimate the expected waiting time at the bus stop

$$\hat{W} = \frac{\overline{N}}{\overline{\lambda}}$$

where

- $\overline{N}$ is the average number of passengers waiting at a stop
- $\overline{\lambda}$ is the average passenger arrival rate

As discussed above, it is fair to conclude that utilisation decreases as number of buses increase. This is beneficial for the passengers, who has a lot of freedom regarding to when and where they want to travel, and gets a more positive experience because of this. For the bus company however, this is a more costly alternative. More buses does not only mean less passengers, but also more fuel and more drivers.

When selecting the number of buses in use, the bus company has to consider passenger satisfactory versus their own operational costs. Fewer buses leads to more passengers on each bus, longer passenger waiting time, leading to less satisfied passengers - but also lower costs. In real life we

would also see an increase the dropout rate, but this in not implemented in this system. More buses gives the opposite result, happier passengers and more costs for the bus system.

II.A.5 Implementing the Passengers

The main difference in the new implementation compared to the old, is how the passengers are represented. Now the passengers are an entity of their own, which gives us an additional measurement, passenger travel time, that we can use to optimise our system further. The implementation of passengers led to a few general changes in the code, such as changing the bus logic, handling the tracking of travel times for each passenger, and some general improvements in the code.

Previous, `Simpy.Container`s were used to keep track of the passenger, but since `Container` does not implement a queuing mechanism, and the `.get()` method picks a random passenger, I switched to `Simpy.Store`. A `Store` object acts as a queue, which is what we want, to ensure that passengers board the bus in the order of arrival.

To track passenger travel times, each passenger record the time they arrive at a stop, the time they board the bus, and the time they leave. When a passenger leaves, the total travel time is calculated and stored in a list for that bus. The overall average travel time and standard error is calculated in the same way as the utilisation described earlier.

This simulation uses `yield` and `process` to synchronise the interaction between passengers and buses. When arriving at a stop, the passenger waits at a stop until a bus arrives and let them board. Likewise, a bus arrives at a stop, and does not leave until the passengers are done boarding. Because of this, we avoid the bus leaving before it is supposed.

```
For 5 buses:
Average utilisation across all simulations: 0.301
Standard error for utilisation: 0.007
Average travel time across all simulations: 17.4
Standard error for travel: 0.757
For 7 buses:
Average utilisation across all simulations: 0.234
Standard error for utilisation: 0.006
Average travel time across all simulations: 15.9
Standard error for travel: 0.555
For 10 buses,:
Average utilisation across all simulations: 0.172
Standard error for utilisation: 0.003
Average travel time across all simulations: 13.4
Standard error for travel: 0.445
```

```
For 15 buses:
Average utilisation across all simulations: 0.125
Standard error for utilisation: 0.001
Average travel time across all simulations: 11.4
Standard error for travel: 0.194
```

Listing 3: Output from extended model and $n_b = [5, 7, 10, 15]$

Listing 3 shows the output after implementing the passenger entity in the simulation. We can see that the utilisation is similar to the previous simulation, which makes sense since the route selection mechanism is unchanged. However, now we have a measurement for passenger travel time as well, which can be used to implement a different selection mechanism.

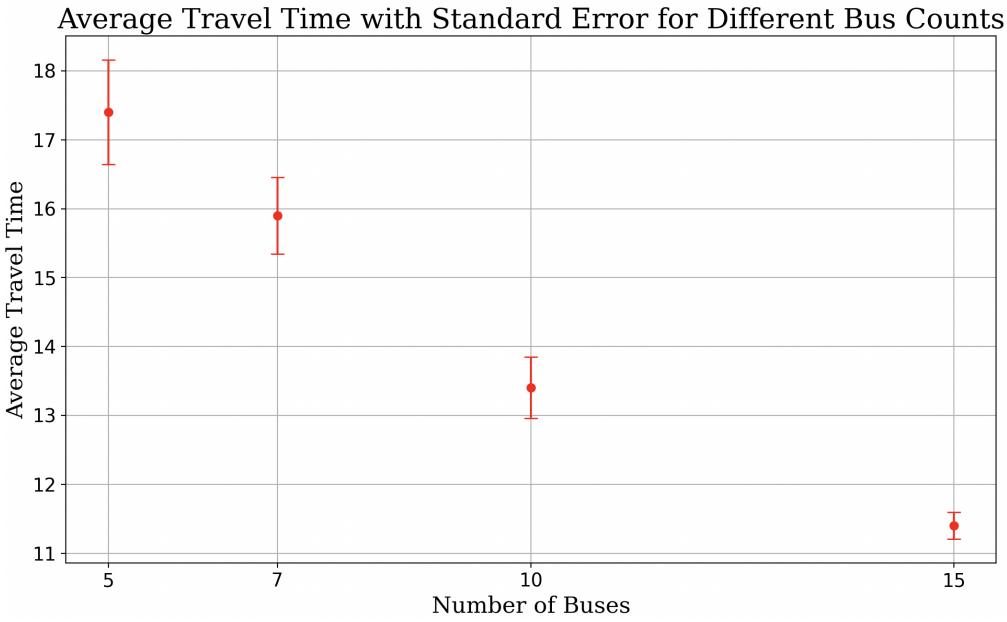


Figure 4: Plot of Average Travel Time and Standard Error

As you can see in Figure 4, the average passenger travel time decreases as the number of busses increase. This is expected, as more buses leads to less passenger waiting time and faster overall travel time.

The implementation of the extended model is available in Appendix D. Additionally, the DES Model the implementation is based on, is available in Appendix F.

Task II.B: Parameter sensitivity

II.B.1 Arrival Rate Sensitivity

The passenger arrival rate measures how often a passenger is generated at a stop. A higher arrival rate, leads to a higher demand for bus. If the number of buses does not satisfy the demand, this will impact the utilisation and the passenger travel time.

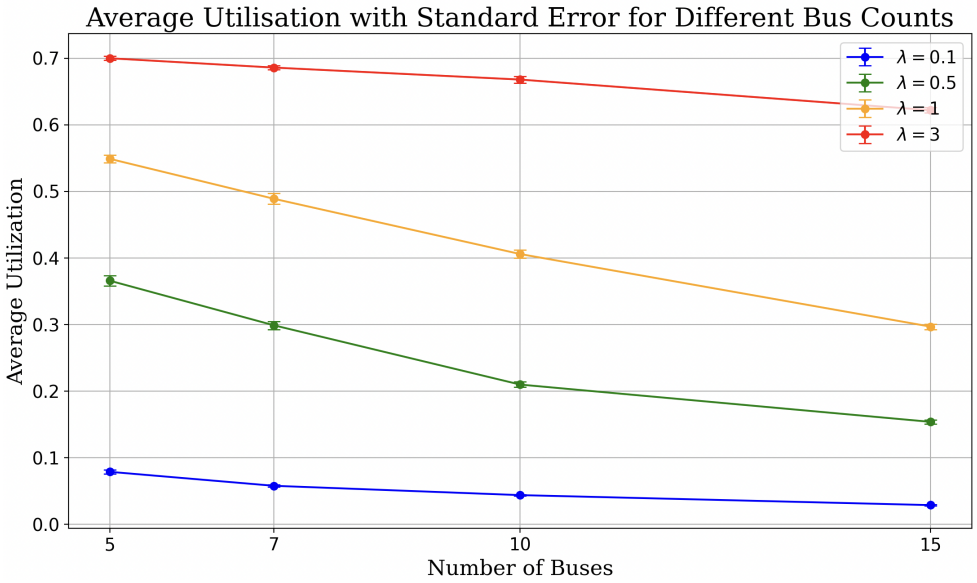


Figure 5: Plot of Average Utilisation with different λ values

Figure 5 shows how bus utilisation changes when we adjust the value of λ. Most noticeably, as λ increases, the utilisation follows. As discussed previously, the utilisation decreases when the number of buses increases. However, for extreme values of λ, the decrease in utilisation is not as significant as for the middle values. This can be due to the fact that, at very low λ, buses have a very low utilisation no matter of the passenger count, and at very high λ, the buses are nearly fully utilised across all bus counts, which makes it harder to distinguish the differences between them.

Figure 6 illustrates how the average passenger travel time is affected by different values of λ. As λ increases, the average travel time also increases, especially when there are fewer buses available. This is expected, since higher λ values indicate more passengers arriving at bus stops, leading to longer waiting times for passengers as buses become more crowded. However, as the number of buses increases, the travel time decreases across all λ values, showing how well the system handles a higher passenger load. We can also see that the travel time remains stable and short for lower values of λ, and that the number of buses has small impact on the travel time. This can be explained by the fact that there is so few passengers, that the there is always room for passengers, even at a low bus count.

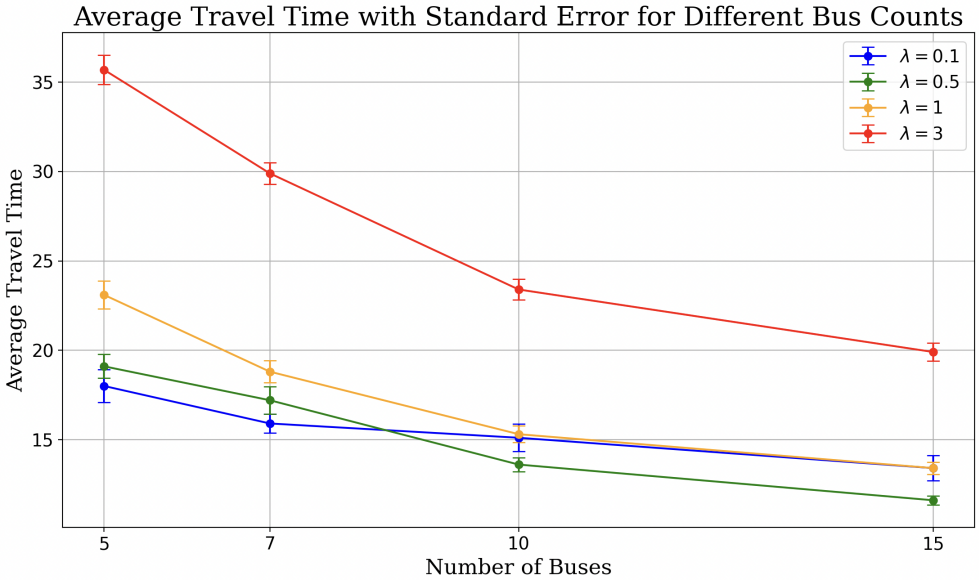


Figure 6: Plot of Average Travel Time with different λ values

II.B.2 Alternative Route Selection

Up until now, the simulation has used a random route selection strategy. This is not ideal as the buses does not consider the conditions on the different routes. I implemented a rule that makes the bus choose the route with the most waiting passengers throughout the route. When running the simulation with the new implementations, the utilisation actually decreased a little bit from the previous simulations. This turned out to be due to the fact that all the buses chose the same route. To fix this I added another rule, saying that there can only be a fixed amount of buses on the same route at any given time. This improved the utilisation significantly. In Figure 7 you can see the result of the new simulation compared to the old one.

Figure 7: Plot of Average Utilisation and Standard Error. Here $max_{\text{bus count}} = 2$.

Figure 8 shows the new average travel time compared to the old one, and as you can see, the new mechanism led to an increase in the travel time. With the new rule, the bus considers both passenger demand and availability of the route. The increase in passenger travel time, can be a result of buses prioritising more popular routes, leading to passengers waiting for a longer amount time before the bus arrives at a stop of less attractive routes.

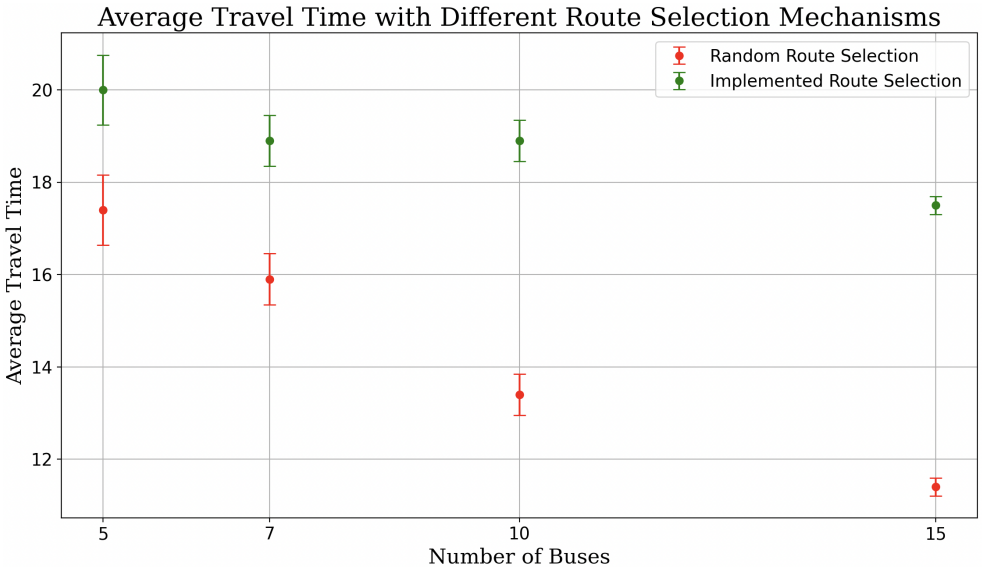


Figure 8: Plot of Average Travel Time and Standard Error. Here $max_{\text{bus count}} = 2$

The "fixed amount" rule needed me to implement a queueing system for situations when number of buses and the restriction do not align. The queue system works in the same way as the bus stop works for the passengers. When a bus has completed its route, it tries to get a new route. If this is not possible, it gets in a queue for idle buses. This queue is updated every time a bus has completed a route.

I tried simulating with a max limit of 1, 2, 3, and 4 buses, to try to find the ideal amount of operational buses. As you can see in Figure 9 the utilisation is relatively independent of how many buses are allowed on the same route. The utilisation decreases with a growing number of buses, as we have seen in previous simulations.

The travel time however, visualised in Figure 10, seems to be more affected by the bus restriction. For lower restrictions, the travel time decreases significantly as the number of buses increases. This can be due to the fact that there always has to be a bus on every route, making sure every stop is served. For a high number of buses, this leads to many idle buses, which is unnecessary, but neither effects the utilisation or the travel time. For higher restrictions, the travel time is not affected as much. This is a probably a result of the buses choosing the same routes, leading to longer passenger times on the less popular routes.

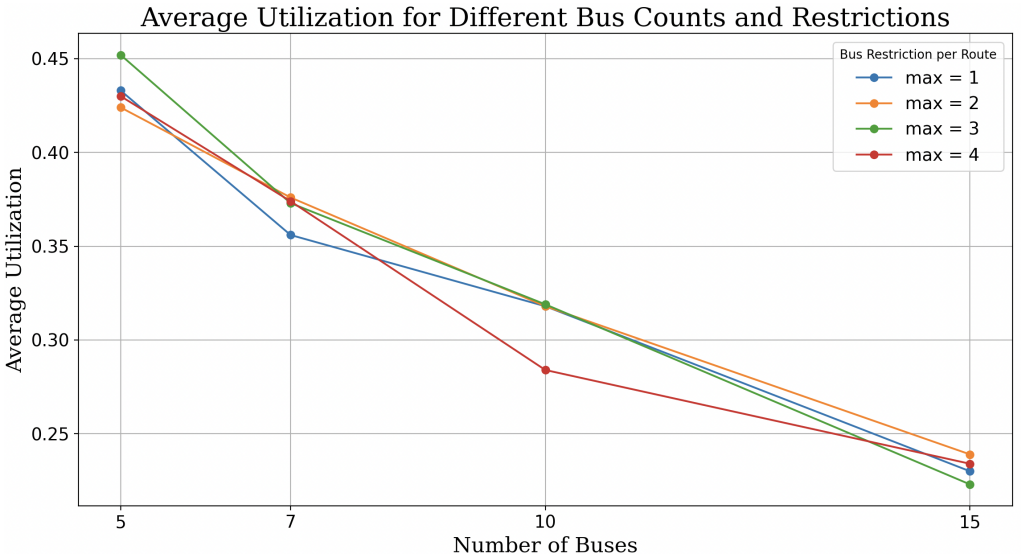


Figure 9: Plot Average Utilisation for Different Bus Counts and Bus Restrictions

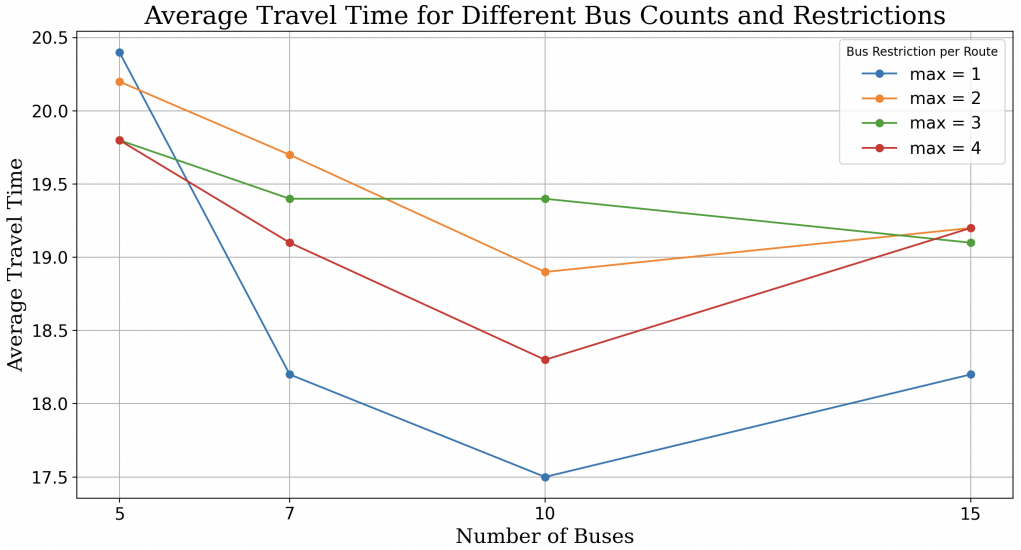


Figure 10: Plot Average Travel Time for Different Bus Counts and Bus Restrictions

While the random route selection led to a slightly better travel time for the passengers, the structured route selection gave a significant increase in the utilisation. This suggests that the right choice between these selection mechanisms depends on the goal of the simulation, to minimise passenger travel time or maximise utilisation.

After testing multiple scenarios, comparing the different utilisation and travel times, and with the given amount of routes in this simulation, I would recommend the bus company to have 7 or 8 buses in use. This takes into account that there is a set limit on the number of buses that can be on the same route at the same time. Considering the results from this simulation and my implementation of the system, I would set that limit to be $max_{\text{bus count}} = 1$. After analysing Figure 9 and Figure 10, I believe that having 7 or 8 operational buses, introduces the optimal balance between bus

utilisation and operational costs, and travel time and passenger satisfaction.

The new route selection implementation is available in Appendix E.

II.B.3 No Pre-Defined Routes

Given a change in the city bus system that leads to fixed bus stops, but no pre-defined routes, we have two alternative scenarios. Either we assume that a passenger knows where they are going and that the bus only let passengers that is going in the same direction or to the same destination to enter the bus, or that passengers still does not have a final destination and the bus implements some rules. Examples of rules are "choose the stop with the most passengers" and "choose the stop of the longest waiting passenger(s)".

General for both cases, I would think that the bus company would need to have a quite high amount of operational buses. This is due to the fact that the buses would be driving a lot more determined, and to ensure that as many passengers as possible are served, the number of buses in use would need to be higher than what has been discussed previously.

In the first scenario I assume that the utilisation, or the general gain for the bus company, would be lower than if the system had pre-defined routes. This is because the bus is more dependent on passengers going in the same direction. On the other hand, I expect the passenger travel time to decrease, as the bus is driving more "straightforward". This requires a lot of operational buses, otherwise the passenger travel time would probably increase as well. All in all, I think this is a bad strategy for the bus company, as the operational costs probably would not outweigh the passenger satisfaction.

In the second scenario, I think both utilisation and average travel time would increase. Since the bus prioritises the more popular stops, the bus will be full more often. However, this also results in more passengers waiting both at the popular stops and at the less popular stops since those stops are not prioritised. The higher travel time, is a trade-off between between higher utilisation and longer waiting time at the stops.

If we were the to remove the fixed bus-stops as well, and make the buses act more like a taxi service, we once again have to consider the costs versus the passenger satisfaction. As discussed above, I think this implementation would result in the need for a rather high bus count to meet the passenger demand. This is because I think it would be hard to synchronise separate passengers as they probably do not have the same origin or destination. Therefore I think a solution like this would be quite costly for the bus company, if they were to take passenger travel time and satisfactory into consideration. After all, if we were to compare this to real life, taxi services tend to be a lot more expensive than regular buses.

Task II.C: Reflection

II.C.1 Observations

One of my main observations from this performance study is the significant impact of different route selection mechanisms on the system's performance. I was surprised by the big difference in utilisation, and considering we only had two routes to choose between, I am convinced it is even more important in cases with more routes. The random route selection gave the best passenger travel time, while the structured route selection improved the route selection significantly at the cost of a slightly higher passenger travel time.

Another observation is the importance of finding an optimal number of operational buses. As the number of buses increases, travel time decreases, but only to a certain extent. At a point, the reduction in travel time decreases to much, that it may not be worth the extra operational costs. This creates a trade-off between reducing passenger waiting times and company costs.

In conclusion, what benefits the bus company the most in terms of efficiency and cost savings may not always align with what is best for passengers when it comes to reducing travel time and improving satisfaction. Optimising the system requires a careful balance between these two factors, ensuring that passenger experience is prioritised without compromizing operational efficiency.

II.C.2 Learning Outcome

This assignment turned out to be very interesting. I probably used too much time focusing on the code, but I got very invested and eager to improve the system as much as possible. I found it difficult to find the optimal balance between passenger travel time and bus utilisation, by altering the number of buses and changing the route selection mechanisms, but I guess that makes sense considering it is the goal of the lab.

On a more practical level, I have spent a lot of time debugging, and fixing A LOT of details - details that probably will go unnoticed by the reader. This has been a very educational experience, but also extremely time-consuming. Implementing a system like this and optimising it was a lot harder than I expected. There is always something you can improve, and while I am pleased with the result, I also have many features I wish I had the time to implement. I really like the objective of the lab, since it is a very realistic problem, and I found it very interesting using simulation to analyse a problem like this.

II.C.3 AI Declaration

The AI tools I have been using in this lab is ChatGPT and GitHub Copilot. I mostly used ChatGPT to help me with the report. I used it to improve some weak paragraphs and to help me with some of the formatting in LaTeX. When I found some of the tasks somewhat unclear, I used AI to help me understand what was expected of me for that task. Despite some flaws, ChatGPT was also useful when I was deep in some heavy debugging. Copilot was mostly used as autocomplete for basic sections of code.

Bibliography

Emstad, Peder J. et al. (2018). *Dependability and Performence with Discrete Event Simulation.* Department of Information Security and Communication Technology at NTNU.

Appendix

A Passenger Distribution Code

```
import matplotlib.pyplot as plt
import numpy as np

mean_interarrival_time = 2
lambda_ = 1 / mean_interarrival_time
n_samples = 1000

def generate_exp_samples(mean_interarrival_time, n_samples):
    u = np.random.uniform(0, 1, n_samples)
    return - mean_interarrival_time * np.log(u)

X_samples = generate_exp_samples(mean_interarrival_time, n_samples)
Y_samples = np.random.exponential(mean_interarrival_time, n_samples)

X_sorted = np.sort(X_samples)
Y_sorted = np.sort(Y_samples)

plt.plot(X_sorted, Y_sorted, marker='o', linestyle='none', color='red')
plt.xlabel('X(j)', size=17, fontdict={'family': 'serif'})
plt.ylabel('Y(j)', size=17, fontdict={'family': 'serif'})
plt.title('Plot of X(j) and Y(j)', size=21, fontdict={'family': 'serif'})
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.legend()
plt.grid()
plt.show()
```

B Simulation Parameters and Route Representation

param.py

```
n_buses = 10 # number of buses
n_buses_list = [5, 7, 10, 15] # for utilisation comparison
n_simulations = 15 # number of simulations
c = 20 # number of seats in each bus
q = 0.3 # probability of a passenger leaving at a stop
max_bus_count = 2 # maximum number of buses per route

travel_times = {
    'R 1': 3, 'R 2': 7, 'R 3': 6,
    'R 4': 1, 'R 5': 4, 'R 6': 3,
    'R 7': 9, 'R 8': 1, 'R 9': 3,
    'R 10': 8, 'R 11': 8, 'R 12': 5,
    'R 13': 6, 'R 14': 2, 'R 15': 3
}

passenger_arrival_rates = {
    'S 1 e': 0.3, 'S 1 w': 0.6,
    'S 2 e': 0.1, 'S 2 w': 0.1,
    'S 3 e': 0.3, 'S 3 w': 0.9,
    'S 4 e': 0.2, 'S 4 w': 0.5,
    'S 5 e': 0.6, 'S 5 w': 0.4,
    'S 6 e': 0.6, 'S 6 w': 0.4,
    'S 7 e': 0.6, 'S 7 w': 0.4
}

routes = {
    "1 eastbound": {
        "S": ["E 1", "S 1 e", "S 4 e", "S 6 e", "E 3"],
        "R": ["R 1", "R 5", "R 8", "R 13"]
    },
    "1 westbound": {
        "S": ["E 3", "S 6 w", "S 4 w", "S 1 w", "E 1"],
        "R": ["R 13", "R 8", "R 5", "R 1"]
    },
    "2 eastbound": {
```

```
        "S": ["E 1", "S 1 e", "S 4 e", "S 7 e", "E 4"],
        "R": ["R 1", "R 5", "R 10", "R 15"]
    },
    "2 westbound": {
        "S": ["E 4", "S 7 w", "S 4 w", "S 1 w", "E 1",],
        "R": ["R 15", "R 10", "R 5", "R 1"]
    },
    "3 eastbound": {
        "S": ["E 2", "S 2 e", "S 5 e", "S 6 e", "E 3"],
        "R": ["R 3", "R 7", "R 9", "R 13"]
    },
    "3 westbound": {
        "S": ["E 3", "S 6 w", "S 5 w", "S 2 w", "E 2"],
        "R": ["R 13", "R 9", "R 7", "R 3"]
    },
    "4 eastbound": {
        "S": ["E 2", "S 3 e", "S 7 e", "E 4"],
        "R": ["R 4", "R 12", "R 15"]
    },
    "4 westbound": {
        "S": ["E 4", "S 7 w", "S 3 w", "E 2"],
        "R": ["R 15", "R 12", "R 4"]
    }
}
```

C Simple Model Code

To run the code on your own computer, ensure that you are in the right folder corresponding to the task you wish to inspect, and run the command `python3 simulation.py`. For example:

```
~/.../LAB II/II.A.2-3-4/
$ python3 simulation.py
```

In the actual code files, you can see many print statements that have been commented out if you wish to get more detailed output. I have removed them from the appendix to keep the main focus on the code.

`simulation.py`

```
# Function to generate passengers at random bus stops
def passenger_generator(env, passenger_container, passenger_arrival_rate):
    while True:
        yield env.timeout(np.random.exponential(1 / passenger_arrival_rate))
        passenger_container.put(1)

# Function for the simulation
def simulation(n_buses, n_simulations):
    utils = []
    for i in range(n_simulations):
        env = sp.Environment()

        all_stops = set()
        for route in param.routes.values():
            stops_in_route = route['S']
            for stop in stops_in_route:
                all_stops.add(stop)

        passenger_containers = {}
        for stop in all_stops:
            passenger_containers[stop] = sp.Container(env, capacity=float('inf'),
            ↪  init=0)

        util = {}
        for bus_id in range(n_buses):
            util[bus_id] = []
```

```
        passenger_arrival_rates = param.passenger_arrival_rates
        for stop, rate in passenger_arrival_rates.items():
            env.process(passenger_generator(env, passenger_containers[stop],
            ↪  rate))

        # Start bus processes
        for id in range(n_buses):
            bus = Bus(env, param.routes, choose_init_route(),
            ↪  passenger_containers, util, id)
            env.process(bus.run())

        env.run(until=100)

        # Calculate average util across all buses in current simulation
        avg_util = np.mean([np.mean(util[id]) for id in util if util[id]])
        utils.append(avg_util)

    print(f'For {n_buses} buses:')
    print(f'Average utilization across all simulations: {round(np.mean(utils),
    ↪  3)}')
    print(f'Standard error: {round(np.std(utils) / np.sqrt(n_simulations), 3)}
    ↪  \n')

for i in range(len(param.n_buses_list)):
    simulation(param.n_buses_list[i], param.n_simulations)
```

bus.py

```
# Route selectors
def choose_init_route():
    return np.random.choice(list(param.routes.keys()))

def choose_next_route(current_route, routes):
    last_stop = routes[current_route]['S'][-1]
    next_routes = []
    for route in routes:
        if routes[route]['S'][0] == last_stop:
            next_routes.append(route)
    chosen_route = np.random.choice(next_routes)
```

```
    return chosen_route

class Bus:
    def __init__(self, env, routes, current_route, passengers, util, id):
        self.env = env
        self.routes = routes
        self.current_route = current_route
        self.passengers = passengers
        self.util = util
        self.id = id
        self.current_passengers = []

    # Function that simulates the "life cycle" of a bus
    def run(self):
        while True:
            stops = self.routes[self.current_route]['S']
            roads = self.routes[self.current_route]['R']
            # Loop through stops and roads
            for stop, road in zip(stops, roads):
                # Passengers leave the bus with probability q
                disembarking_passengers = []
                for passenger in self.current_passengers[:]:
                    if np.random.rand() < param.q:
                        disembarking_passengers.append(passenger)
                for passenger in disembarking_passengers:
                    self.current_passengers.remove(passenger)

                # Passengers board the bus
                passengers_to_pickup = min(self.passengers[stop].level, param.c -
                ↪  len(self.current_passengers))
                if passengers_to_pickup > 0:
                    self.current_passengers += [1] * passengers_to_pickup
                    yield self.passengers[stop].get(passengers_to_pickup)

                # Calculcate new utilisation
                current_util = len(self.current_passengers) / param.c
                self.util[self.id].append(current_util)

                # Travel to next stop
```

```
        travel_time = param.travel_times[road]
        yield self.env.timeout(travel_time)

    # Reset
    self.current_passengers = []
    self.current_route = choose_next_route(self.current_route,
    ↪  self.routes)
```

D Implementation of Passenger Entity

simulation.py

```
def simulation(n_buses, n_simulations):
    utils = []
    passenger_travel_times = []
    for _ in range(n_simulations):
        passenger_id = 0
        env = sp.Environment()

        all_stops = set()
        for route in param.routes.values():
            stops_in_route = route['S']
            for stop in stops_in_route:
                all_stops.add(stop)

        waiting_passengers = {}
        for stop in all_stops:
            waiting_passengers[stop] = sp.Store(env)

        util = {}
        for bus_id in range(n_buses):
            util[bus_id] = []

        travel_time = {}
        for bus_id in range(n_buses):
            travel_time[bus_id] = []

        passenger_arrival_rates = param.passenger_arrival_rates
        for stop, rate in passenger_arrival_rates.items():
            env.process(passenger_generator(passenger_id, env, str(stop),
            ↪  waiting_passengers[str(stop)], rate))

        # Start bus processes
        for bus_id in range(n_buses):
            bus = Bus(env, param.routes, choose_init_route(), waiting_passengers,
            ↪  util, bus_id, travel_time)
            env.process(bus.run())
```

```
        env.run(until=100)

        # Calculate average util and travel time across all buses in current
        ↪ simulation
        avg_util = np.mean([np.mean(util[bus_id]) for bus_id in util if
        ↪ util[bus_id]])
        utils.append(avg_util)
        avg_travel_time = np.mean([np.mean(travel_time[bus_id]) for bus_id in
        ↪ travel_time if travel_time[bus_id]])
        passenger_travel_times.append(avg_travel_time)

    print(f'For {n_buses} buses:')
    print(f'Average utilisation across all simulations: {round(np.mean(utils),
    ↪ 3)}')
    print(f'Standard error for utilisation: {round(np.std(utils) /
    ↪ np.sqrt(n_simulations), 3)}')
    print(f'Average travel time accross all simulations:
    ↪ {round(np.mean(passenger_travel_times), 1)}')
    print(f'Standard error for travel: {round(np.std(passenger_travel_times) /
    ↪ np.sqrt(n_simulations), 3)} \n')

for i in range(len(param.n_buses_list)):
    simulation(param.n_buses_list[i], param.n_simulations)
```

bus.py

```
# Route selectors
def choose_init_route():
    chosen_route = np.random.choice(list(param.routes.keys()))
    return chosen_route

def choose_next_route(current_route, routes):
    last_stop = routes[current_route]['S'][-1]
    next_routes = []
    for route in routes:
        if routes[route]['S'][0] == last_stop:
            next_routes.append(route)
    chosen_route = np.random.choice(next_routes)
    return chosen_route
```

```
class Bus:
    def __init__(self, env, routes, current_route, waiting_passengers, util, id,
    ↪  passenger_travel_times):
        self.env = env
        self.routes = routes
        self.current_route = current_route
        self.waiting_passengers = waiting_passengers
        self.util = util
        self.id = id
        self.current_passengers = []
        self.passenger_travel_times = passenger_travel_times

    def disembark(self):
        disembarking_passengers = []
        for passenger in self.current_passengers[:]:
            if np.random.rand() < param.q:
                disembarking_passengers.append(passenger)
        for passenger in disembarking_passengers:
            time = passenger.disembark()
            self.passenger_travel_times[self.id].append(time)
            passenger.disembark()
            self.current_passengers.remove(passenger)
        return len(disembarking_passengers)

    def embark(self, stop):
        waiting_passengers = len(self.waiting_passengers[stop].items)
        embarking_passengers = min(waiting_passengers, param.c -
        ↪  len(self.current_passengers))
        if embarking_passengers > 0:
            for _ in range(embarking_passengers):
                passenger = yield self.waiting_passengers[stop].get()
                self.current_passengers.append(passenger)
                passenger.board()
        return embarking_passengers

    def run(self):
        while True:
            stops = self.routes[self.current_route]['S']
```

```
            roads = self.routes[self.current_route]['R']
            # Loop through stops and roads
            for stop, road in zip(stops, roads):
                # Passengers leave the bus with probability q
                disembarking_passengers = self.disembark()

                # Passengers board the bus
                embarking_passengers = yield self.env.process(self.embark(stop))

                # Calculate util
                current_util = len(self.current_passengers) / param.c
                self.util[self.id].append(current_util)

                # Travel to next stop
                travel_time = param.travel_times[road]
                yield self.env.timeout(travel_time)

            self.current_passengers = []
            self.current_route = choose_next_route(self.current_route,
            ↪  self.routes)
```

passenger.py

```
def passenger_generator(id, env, start_stop, container, passenger_arrival_rate):
    while True:
        yield env.timeout(np.random.exponential(1 / passenger_arrival_rate))
        passenger = Passenger(env, id, start_stop, container)
        id += 1
        env.process(passenger.wait())

class Passenger:

    def __init__(self, env, id, start_stop, container):
        self.env = env
        self.id = id
        self.start_stop = start_stop
        self.container = container
        self.arrival_time = 0
```

```
        self.board_time = 0
        self.disembark_time = 0
        self.total_time = 0

    def wait(self):
        self.arrival_time = self.env.now
        yield self.container.put(self)

    def board(self):
        self.board_time = self.env.now

    def disembark(self):
        self.disembark_time = self.env.now
        self.total_time = self.disembark_time - self.arrival_time
        return self.total_time

    def __str__(self):
        return f'ID: {self.id} start_stop: {self.start_stop} Arrival time:
        ↪ {self.arrival_time}, Board time: {self.board_time}, Disembark time:
        ↪ {self.disembark_time}, Travel time: {self.total_time}'
```

E Implementation of Route Selection

simulation.py

```
def simulation(n_buses, n_simulations):
    utils = []
    passenger_travel_times = []
    for _ in range(n_simulations):
        passenger_id = 0
        env = sp.Environment()

        all_stops = set()
        for route in param.routes.values():
            stops_in_route = route['S']
            for stop in stops_in_route:
                all_stops.add(stop)

        waiting_passengers = {}
        for stop in all_stops:
            waiting_passengers[stop] = sp.Store(env)

        util = {}
        for bus_id in range(n_buses):
            util[bus_id] = []

        travel_time = {}
        for bus_id in range(n_buses):
            travel_time[bus_id] = []

        bus_count_per_route = {route: 0 for route in param.routes.keys()}

        idle_queue = sp.Store(env)

        # Start passenger generators for each stop with their respective rates
        passenger_arrival_rates = param.passenger_arrival_rates
        for stop, rate in passenger_arrival_rates.items():
            env.process(passenger_generator(passenger_id, env, str(stop),
            ↪  waiting_passengers[str(stop)], rate))
```

```
        # Start bus processes
        for bus_id in range(n_buses):
            bus = Bus(env, param.routes, choose_init_route(), waiting_passengers,
            ↪  util, bus_id, travel_time)
            env.process(bus.run(bus_count_per_route, idle_queue))
            bus_count_per_route[bus.current_route] += 1

        env.run(until=100)

        # Calculate average utilization and travel time across all buses in this
        ↪  simulation
        avg_util = np.mean([np.mean(util[bus_id]) for bus_id in util if
        ↪  util[bus_id]])
        utils.append(avg_util)
        avg_travel_time = np.mean([np.mean(travel_time[bus_id]) for bus_id in
        ↪  travel_time if travel_time[bus_id]])
        passenger_travel_times.append(avg_travel_time)

    print(f'For {n_buses} buses:')
    print(f'Average utilisation across all simulations: {round(np.mean(utils),
    ↪  3)}')
    print(f'Standard error for utilisation: {round(np.std(utils) /
    ↪  np.sqrt(n_simulations), 3)}')
    print(f'Average travel time accross all simulations:
    ↪  {round(np.mean(passenger_travel_times), 1)}')
    print(f'Standard error for travel: {round(np.std(passenger_travel_times) /
    ↪  np.sqrt(n_simulations), 3)} \n')

for i in range(len(param.n_buses_list)):
    simulation(param.n_buses_list[i], param.n_simulations)
```

bus.py

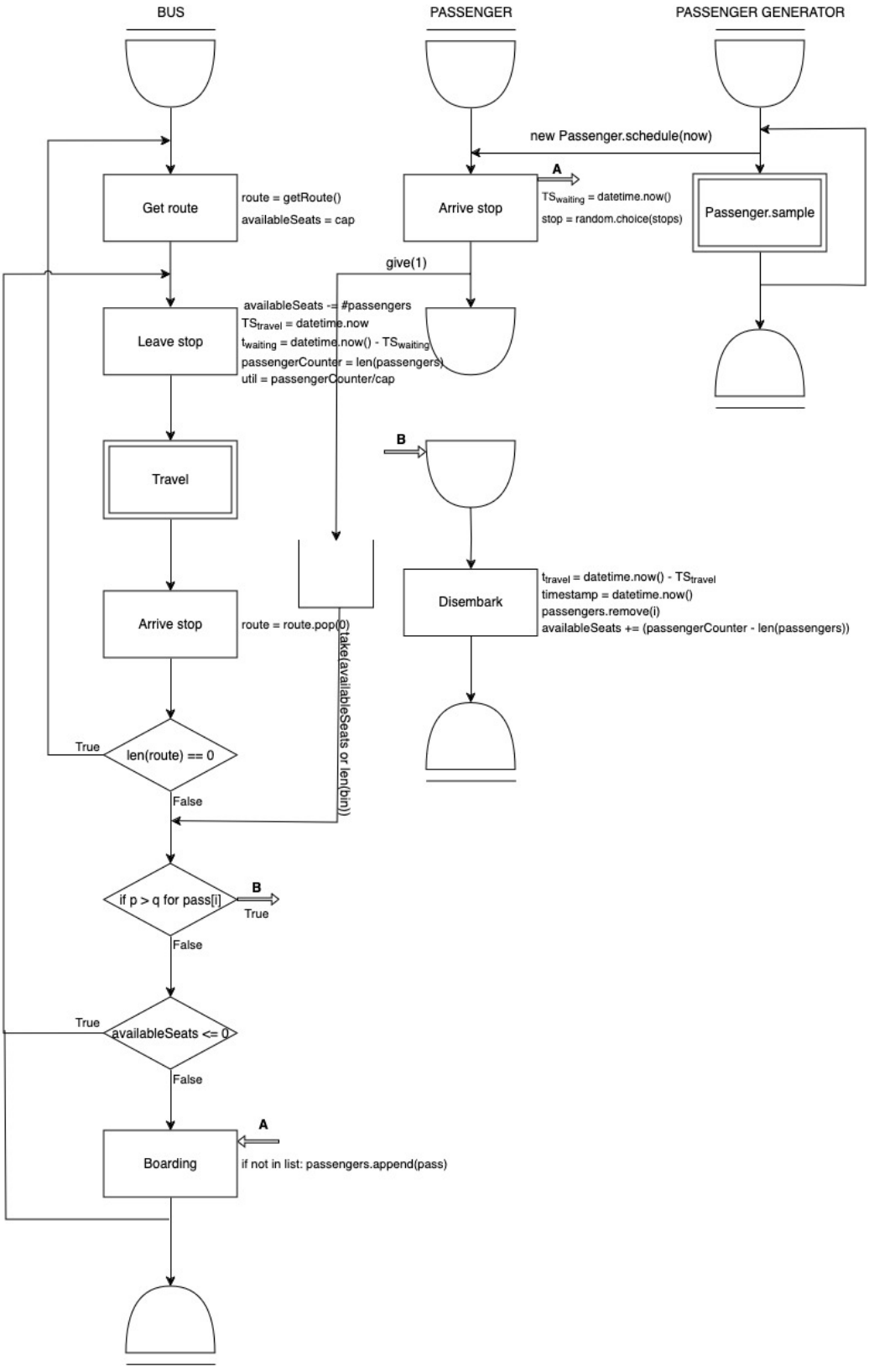
```
# Function to choose intital route
def choose_init_route():
    return np.random.choice(list(param.routes.keys()))

# Function to choose next route when at last stop. Chooses the route with the
↪  most passengers waiting
```

```
def choose_next_route(current_route, routes, waiting_passengers,
↪  bus_count_per_route):
    last_stop = routes[current_route]['S'][-1]
    potential_routes = []

    # Find routes that start at the last stop of the current route and have less
    ↪  than x buses
    for route in routes:
        if routes[route]['S'][0] == last_stop and bus_count_per_route[route] <
        ↪  param.max_bus_count:
            potential_routes.append(route)

    # Choose the route with the most passengers waiting at all stops
    best_route = None
    max_passengers = 0
    for route in potential_routes:
        total_passengers = sum(len(waiting_passengers[stop].items) for stop in
        ↪  routes[route]['S'])
        if total_passengers > max_passengers:
            max_passengers = total_passengers
            best_route = route

    return best_route

class Bus:
    def __init__(self, env, routes, current_route, waiting_passengers, util, id,
    ↪  passenger_travel_times):
        self.env = env
        self.routes = routes
        self.current_route = current_route
        self.waiting_passengers = waiting_passengers
        self.util = util
        self.id = id
        self.current_passengers = []
        self.passenger_travel_times = passenger_travel_times

    def disembark(self):
        disembarking_passengers = []
        for passenger in self.current_passengers[:]:
```

```
            if np.random.rand() < param.q:
                disembarking_passengers.append(passenger)
        for passenger in disembarking_passengers:
            time = passenger.disembark()
            self.passenger_travel_times[self.id].append(time)
            passenger.disembark()
            self.current_passengers.remove(passenger)
        return len(disembarking_passengers)

    def embark(self, stop):
        waiting_passengers = len(self.waiting_passengers[stop].items)
        embarking_passengers = min(waiting_passengers, param.c -
        ↪  len(self.current_passengers))
        if embarking_passengers > 0:
            for _ in range(embarking_passengers):
                passenger = yield self.waiting_passengers[stop].get()
                self.current_passengers.append(passenger)
                passenger.board()
        return embarking_passengers

    def run(self, bus_count_per_route, idle_queue):
        while True:
            stops = self.routes[self.current_route]['S']
            roads = self.routes[self.current_route]['R']

            # Loop through stops and roads
            for stop, road in zip(stops, roads):
                stop_name = stop[0] + stop[2]

                # Passengers leave the bus with probability q
                disembarking_passengers = self.disembark()

                # Passengers board the bus
                embarking_passengers = yield self.env.process(self.embark(stop))

                # Calculate util
                current_util = len(self.current_passengers) / param.c
                self.util[self.id].append(current_util)
```

```
        # Travel to next stop
        travel_time = param.travel_times[road]
        yield self.env.timeout(travel_time)

    bus_count_per_route[self.current_route] -= 1
    if len(idle_queue.items) > 0:
        idle_bus = yield idle_queue.get()
        available_route = choose_next_route(self.current_route,
        ↪  self.routes, self.waiting_passengers, bus_count_per_route)
        if available_route:
            bus_count_per_route[available_route] += 1
            idle_bus.current_route = available_route
            self.env.process(idle_bus.run(bus_count_per_route,
            ↪  idle_queue))
```

F DES Model


BUS
PASSENGER
PASSENGER GENERATOR
new Passenger.schedule(now)
Get route
route = getRoute()
availableSeats = cap
Arrive stop
A
TS_waiting = datetime.now()
stop = random.choice(stops)
Passenger.sample
give(1)
Leave stop
availableSeats -= #passengers
TS_travel = datetime.now
t_waiting = datetime.now() - TS_waiting
passengerCounter = len(passengers)
util = passengerCounter/cap
Travel
B
Disembark
t_travel = datetime.now() - TS_travel
timestamp = datetime.now()
passengers.remove(i)
availableSeats += (passengerCounter - len(passengers))
Arrive stop
route = route.pop(0)
take(availableSeats or len(bin))
True
len(route) == 0
False
if p > q for pass[i]
B
True
False
True
availableSeats <= 0
False
A
Boarding
if not in list: passengers.append(pass)