Eindhoven University of Technology
Department of Mathematics & Computer Science

Queueing at Waste Recycling Plants

Thomas van Kaathoven Waseem Khan David Meijll

Contents

1 Introduction

In recent times, the storage and processing of waste has grown to become a large problem in our society. Even though man has always been a littering species, the rise of consumerism and mass production of goods has made this a larger challenge to solve, with multiple negative side effects. For instance, the space required to store or recycle different types of waste can no longer be used for anything else, and the transport and processing of waste produce harmful substances that impact our health. It is therefore crucial that these waste storages and processors operate with optimal efficiency.
In the case of Eindhoven, there are currently two waste recycling points (WRPs), where citizens of Eindhoven can bring their waste using a city pass, which usually is done by car or van. Although waste production has certainly not decreased, there were formerly three WRPs. One of these was closed down in 2012, being replaced by a new WRP in a different location, with another being closed in 2013.
We now wonder if the WRP closed in 2012 could have been made more efficient, instead of being replaced. After all, the constant flow of cars entering and exiting a WRP can cause queues. These queues can be minimised by optimising the design of the WRP. Specifically in the case of this WRP, the *Gabriël Metsulaan WRP*, any queue longer than three cars requires additional cars to queue on the road, which worries the municipality of Eindhoven.

In this report, we will simulate the WRP as it was in 2012, in order to identify the queue lengths and waiting times of the visiting cars. Following this, we will analyse the performance of the WRP, looking at various numbers of customers. and suggest improvements that increase the efficiency. Doing this, we take into account both the cost and the efficiency of the solution we propose.

2 Model Description

Simulating the WRP will require us to set up a mathematical model that describes all the characteristics of the situation. For this, we need a number of assumptions.
Firstly, we look at the layout of the WRP, based on the real-life situation. We assume that it is divided into 5 sections, corresponding to 4 different types of waste, which are

1. **Entrance**, the gate all cars pass through entering the WRP, which contains one lane.
2. **Flammable** and **Wood/Paper/Metal** (FWPM1&2), with four parking spots,
3. **Debris/Gypsum/Chemicals** (DcDdGpCh), with four parking spots,
4. **Green/Wood/Tires/Electrical/Gas** (GrWcTEGs), with three parking spots,
5. **Rest**, including the exit, with two parking spots.

The order of these sections matters, as it is compulsory to visit them in this order. To know how the WRP is used, we now consider the process of bringing waste to the WRPs. We assume that

- most customers arrive by car or van.
- most customers carry multiple types of waste.
- upon arrival to the gate, customers need to show their city pass, one customer at a time, in order to be granted entry into the WRP.
- if any of the parking lots at waste stations that a customer needs to visit is full, the customer needs to wait at the entrance until the required spots are free.
- customers carrying multiple types of waste, will park once in each corresponding section.
- customers dump their waste on foot.

From these assumptions, we can already see that queueing problems can occur. For instance, customers with a lot of waste in one section will keep their car parked in one spot for a long time.
As mentioned, customers can arrive by car and by van. To distinguish between the two, we make two more assumptions.

- Vans and cars with trailers are both seen as 'Big vehicles'.
- Big vehicles occupy two parking spots, while regular cars occupy one.

Additionally, although it is rare, customers can arrive by bike or on foot, in which case they can move to the front of the queue immediately.
Finally, since we aim to keep the simulation as realistic as possible, there are some necessary assumptions regarding the 'human element' of the WRP.

- Long queues can cause customers to get impatient and leave the queue.
- Customers without a city pass who accidentally drive up to the gate can drive through the WRP without being allowed to dump their waste.
- Customers can arrive to the entrance gate before the WRP opens.

2.1 Data

To aid in accurately simulating the WRP and determining its efficiency, we are given data sets. The first data set contains the description, arrival time and departure time of each car, and also the type of waste the customer brought. The data in this set was collected on days where the WRP was open from 10am to 2pm.
In addition to the data described above, the second data set also contains the arrival time at the queue for each car, whether the gate was blocked or not, and comments indicating any specialties regarding the arrival. This data was collected on September 4, 6, and 7, 2012, when the WRP was open from 1pm to 5pm. Although this data set is more extensive and realistic than the first one, there are some difficulties that arise when using this one. For instance, the service time of a vehicle is not recorded in the data, and instead has to be computed comparing the entrance times (the time a vehicle is granted entry into the WRP) of successive vehicles.
Ultimately, since the second data set is more realistic, we will use it in our simulations. This way, we can give more accurate recommendations to the municipality than we could if we were using the first data set.

3 Implementation

Having set up the model, we can start simulating the WRP and producing results. To do this, we use object-oriented programming in Python.
We use several classes in the implementation of this model, namely `FES`, `Event`, `Vehicle`, `GateQueue`, `Gate`, `GarbageQueue` and `Station`, which we will explain in the following section. Finally, all classes are tied together with the class `Simulation`.

3.1 Main classes

Firstly, for the class `FES`, Future Events Set, we use the Python heapq module, an implementation of the heap queue algorithm, or priority queue algorithm, and add 4 methods. These methods are firstly a heappush to add events, then a heappop to get the next event, thirdly a boolean `isEmpty` function which checks if the number of events is 0, and finally a function that returns the number of events in the heap.

Next, the class `Event` is used to simulate the 4 possible events:

- `ARRIVAL` indicates an arrival to the WRP.
- `STATIONMOVE` indicates a vehicle moving to another waste station.
- `DEPARTURE` indicates a departure from the WRP.
- `IMPATIENT` indicates a customer has gotten impatient and leaves the queue.

In short, these events describe the 'moves' a customer can make regarding the WRP. For each of these, we initiate the event with the vehicle type and time. We use the `lt` method, which acts as a less than-operator, as a comparative for the different events in terms of priority.
Thirdly, we have the class `Vehicle`. The Vehicle can either be a PEDESTRIAN, BIKE, CAR or VAN. Each vehicle is initialised with a number of characteristics, namely its arrival time, waste information, impatience (which denotes the time after which a customer gets impatient), and whether a customer has a city pass or not. We then include several different methods to set the departure time, change to the next station, indicate when there are no stations left and a comparative to determine the priority of each type of vehicle.
Fourthly, we have the class `GateQueue`. This class creates a queue at the gate, and contains a number of method. These are

- `add`: adds a vehicle to the queue;
- `next`: moves to the next vehicle in the queue;
- `isEmpty`: checks whether the queue is empty;
- `length`: finds the length of the queue;
- `remove`: removes a vehicle from the queue.

The next class, `Gate`, is used to simulate the activity at the entrance gate, with the methods `add`, `queueToGate`, `remove`, `isFull`, `fill` that describe the possible actions that can be taken at the entrance.
Similar to the `GateQueue` class, we now introduce the class `GarbageQueue`, with the same methods `add`, `remove`, `isEmpty` and `length`.
Lastly, we have the class `Station`, which is initialised with a name, position in the order, capacity, and service time. This class contains the same methods as the queue-related classes, with the addition of some methods that determine whether a vehicle can enter the WRP based on the capacity of the station. Additionally, the method `genServiceTime` generates a service time at the station, based on the gamma distribution.

3.2 Simulation

Now that we have described all the functions, we come to the class that ties it all together; the `Simulation`. First we initialise all the `Stations` in the compound. We then set run to True and create a while loop with run. Getting the next `Event` from the `FES`, we run through the respective outcomes for each event.

Firstly, for event `ARRIVAL`, we add a `Vehicle` to the `Gate`. We then check if the gate is empty by referencing to `gateQueue`. If the gate is empty, the `Vehicle` plans the gate move. If the `Gate` is not empty, the event IMPATIENT is created and added to the `FES`. We then simulate the next `ARRIVAL` event.
Next, if event is `STATIONMOVE`, we set `oldStation` to the vehicles current station. If `oldStation` $= 0$, i.e., the current station is the entrance gate, we first check if the vehicle can fit in all stations it needs to visit. If so, we look at the next station the vehicle is supposed to go to. We then add the vehicle to the next station and service the vehicle immediately if the station is currently empty. We then remove it from the gate, since it has moved off from there. If there are any other vehicles in the gate queue, we move to the next vehicle and plan the gate move. Next, if not all the cars can fit in the station, we set the next event to be arrival and station to be 0.
If `oldStation` $\neq 0$, i.e., the current station is not the entrance, we will switch the vehicle to the next station, and add the vehicle to the new station. If the new station is empty, we plan a station move. we then update the old station.
Next, if the event type is a departure, we get `oldStation` from the vehicles current station. If `oldStation` is the gate, we remove the vehicle from the gate. If the queue at the gate is not empty, we can get the next vehicle at the gate and plan a gate move. If `oldStation` is not the gate, we update it, and then log the vehicle.
Lastly, if the event type is `IMPATIENT`, and the vehicle is not serviced, we remove the vehicle from the queue at the gate and log the impatience of the vehicle.

At the end of every iteration of the `FES` we log the queues at the end and check the `FES`. Once the `FES` is empty, we set run to false and close the while loop.
To facilitate the function above, we created several helper functions as follows.

- `createArrival`: This function adds a new arrival event to the `FES`. It checks for the correct vehicle arrival density, then checks if the arrivals are sparse. With sparse arrivals, a first arrival may take longer than expected so we adjusted it to the next density after half an hour. We then check if the arr time is less than the closing time; as long as vehicles are allowed to arrive , we can create a new arrival event. We then generate the stations to be visited randomly and add instantiate the vehicle and the event, adding the event to the `FES`.

- `planGateMove`: This function creates the next `STATIONMOVE` event to occur from the gate. We first get the vehicle wait time at the gate and service time and set the serviced status of the car to True. If the vehicle forgot its city pass, we set the vehicle departure time to the service time, otherwise we set the vehicle arrival time to the service time. We then create a station move event and add it to the `FES`.
- `planStationMove`: This function creates the next event to occur from the station. We first check if there are any other stations left for the car to visit. If there are no stations, we plan the departure of the car and create a `DEPARTURE` event, setting the departure time to be the service time. If there are still stations left, we plan a `STATIONMOVE` event, setting the vehicle station arrival time to be the service time and adding the event to the `FES`.
- `updateOldStation`: This function is used to update the old station. We first remove the vehicle from the station, then check if the queue is not empty and if the station queue fits. If so, we get the new vehicle from the station, add the wait time of the vehicle to the station wait time, and run the `planStationMove` function.
- `logQueues`: This function is used to log the queue lengths at each of the stations.
- `logImpatient`: This function is used to log the time, vehicle impatiency time-arrival time and queue length at the gate.
- `logVehicle`: This function is used to log the vehicle type, arrival time, departure time, gate waiting time and station waiting time. We check if the customer remembered his city pass. If forgotten, we indicate this with a 0.

3.3 Obtaining results

Now that we are able to simulate a run, we create several functions to get good results as follows. All code relating to this section can be found in subsection A.2.

- `getData`: this function takes in the filenames and uses pandas to read all the excel file names
- `getArrivalCounts`: with the arrival times, we filter the arrivals in 30 minute intervals.
- `getArrVars`: taking in all the data and the sheets, we get arrival counts per 30 minute intervals as well as first arrival, opening and closing times.
- `getGateServiceVars`: this function gets the gate service times and matches them to a gamma distribution.
- `getStationServiceVars`: this function gets station service times and matches them to a gamma distribution.
- `getGarbageAtStations`: this function gets the garbage collected at each station.
- `getGarbageTypeChance`: this function gets the chance each type of garbage is present
- `getForgetChance`: this function finds 'pass' in the comments of the entrance sheet to see if anyone has forgotten their pass.

- `getImpatiencyVars`: this function tries to find variables for the gamma distribution of the times people are willing to wait in queue
- `getVehicleVars`: this function gets the chances of each type of vehicle showing up.
- `convertTime`: this function converts time from the excel sheet into seconds since the start of the day.

In addition to all the functions described, we create 2 functions. Firstly, `stepPlot`, to plot the queue and occupancy over time, and secondly, `histPlot`, to plot a histogram for the respective values. Now that we have all our functions, we can run the code and get the variables from the files. With our variables ie arrivals, startTime, openTime, closeTime, garbageTypeChanceDict, stations, stationGarbage, gateServiceVars, vehicleVars, forgetChance, stationServiceVars and impatiencyVars, we initialise the simulation. We set up repeated simulations and get the results as described in the next section.

4 Results

Using the classes that were introduced in section 3, we now have everything we need to analyse the performance of the Gabriël Metsulaan WRP.

First, we simulate the waiting times and queue lengths of vehicles in the WRP. The functions described in subsection 3.3 are used to generate and format the data we need to obtain meaningful results. Following this, we input the number of entrances, which is 1, and the order and capacity of each waste station. Then, for 200 runs, we simulate the queue, keeping track of impatient cars as well. In this case, the total time is $3 \cdot 4 \cdot 60 \cdot 60 = 43200$ seconds. Our code concludes that

> "In 200 runs the mean time (s) when the gate queue is too long is 10823.418853171086 ± 1222.80256377635.
> From an average of 135.375 ± 11.745823725903602 vehicles, an average of 97.95 ± 12.67112860008926 left the queue,
> these had an average waiting time (s) of 645.9196009794591 ± 158.16136487491232.
> Vehicles usually spent 814.1980053707666 ± 553.1347993985556 seconds waiting at the gate
> and 6.367748186032243 ± 49.596970769949415 seconds waiting at stations. They also spent an average of 850.172365261323 ± 660.8645204670183 seconds inside the WRP."

where each number is given in seconds. Some interesting things to note are that on average, a large portion of the vehicles leaves the queue because of impatience. This can be explained by the high average waiting time. When the waiting time is over 800 seconds on average, it can be imagined that many customers will be impatient to such a degree that they choose to leave the queue. On the other hand, we see that the waiting times at the stations themselves is low on average, albeit with a relatively high standard deviation. This shows that this method of having a free spot at each necessary station before a customer can enter, does help with the queueing inside of the WRP. However, queueing is only a small part of the total time spent by customers in the WRP, implying that the process of unloading waste and dumping it by foot takes a large amount of time.

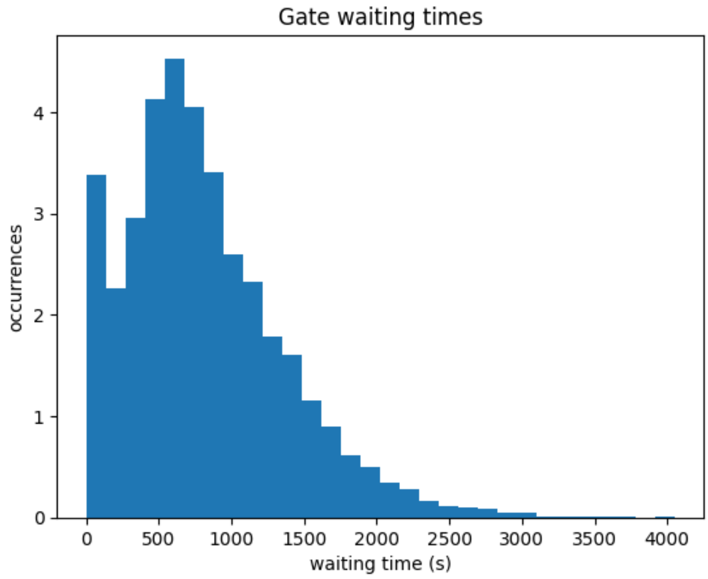


Figure 1: Histogram of waiting times at the gate.

To analyse the waiting times in more depth, Figure 1 shows a histogram of the waiting times at the entrance. We can see that the vast majority of waiting times are close to the average that was found above, and the frequency quickly dropping off for the higher waiting times.

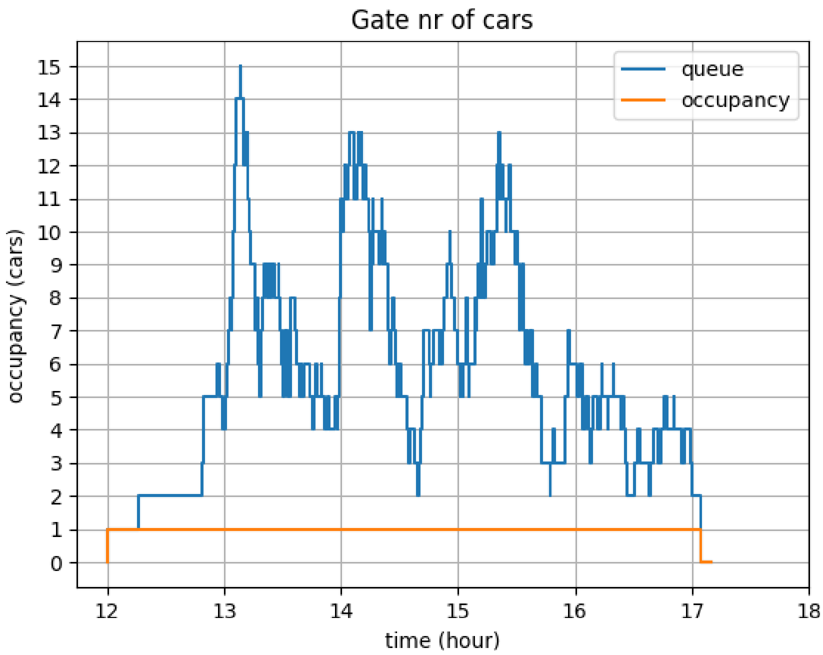
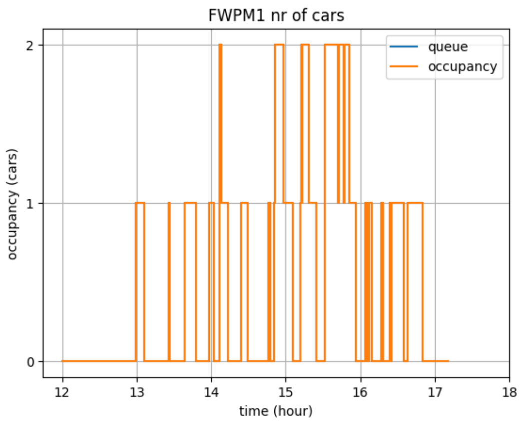
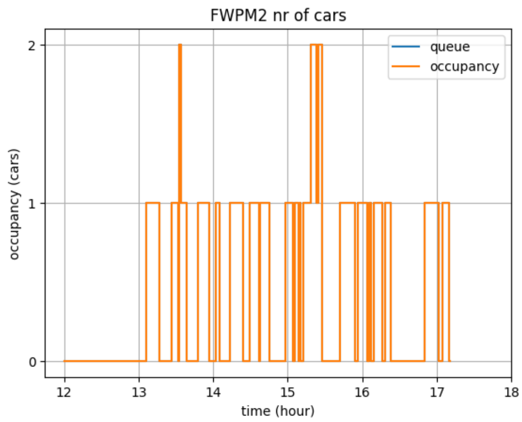


Figure 2: Gate queue size and occupancy over time.

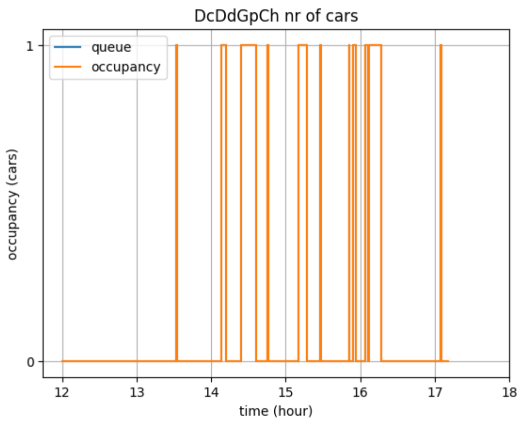
Next, as shown in Figure 2, the queue and occupancy of vehicles at the gate were plotted against time. Note that the occupancy of the entrance is at most one, since there is only one queue, that can service one vehicle at a time. Overall, we see that at most times, the queue length is quite large, especially when taking into account the design flow regarding queue sizes larger than three. Specifically for these runs, there was always at least one car at the gate, either waiting or being serviced, although most cars arrived near the middle of the afternoon.



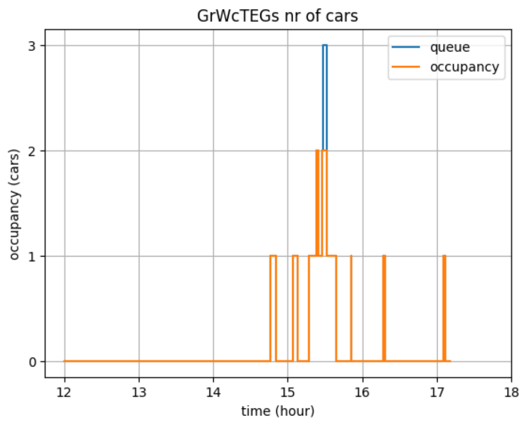
(a) FWPM1



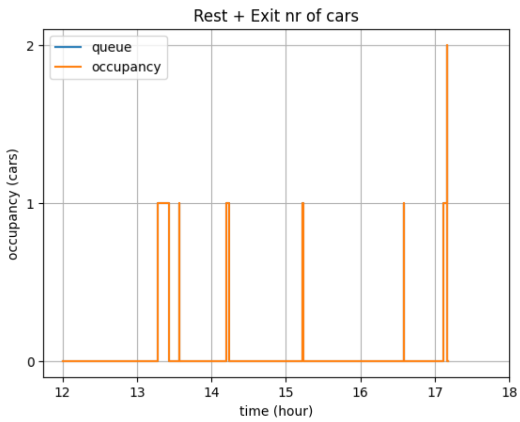
(b) FWPM2



(c) DcDdGpCh



(d) GrWcTEGs



(e) Rest + Exit

Figure 3: Occupancy over time for each type of waste.

Following this, we take a look at the occupancy of each waste station in the WRP. By comparing the smaller figures in Figure 3, we see that both FWPM-stations were occupied often throughout the time period, which is not surprising, since it was established earlier that these stations are by far the most 'popular'. Another interesting thing we see is that a queue occurs at the fourth station, even though this station is not the most visited one. This can however be explained by the fact that it is one of the last stations in the WRP, meaning that the probability that a customer in the WRP still has to visit this station is higher. Hence, running our simulation again could result in this queue appearing elsewhere.

5 Improvements

There are a multitude of different solutions we can implement to reduce the waiting times and queue lengths at the WRP.
Firstly, the queue peaks at certain hours of the day. Spreading awareness of this would enable customers to come at off peak hours, significantly reducing the waiting times of the cars.
Secondly, we can observe that the waiting times at the gate are consistently more than 1 and can go up to 15, while there is no significant queue at the different stations. This bottle neck at the gate needs to be addressed. Several changes include increasing the occupancy limit at the gate to more than 1, and increasing the parking lots at the different stations to accommodate a higher influx of customers. Reducing the servicing times would help make the process faster and reduce the waiting times.
Lastly, an alternative way we can address the bottle neck at the gate is by creating more parking spots so that there will be fewer cars on the road and the performance will be improved; one of the measures is the fraction of time that the queue is longer than 3 vehicles,the number of parking spots, and with more parking spots this fraction will be significantly reduced.

6 Conclusion

In this report, we sought out to simulate the Gabriël Metsulaan waste recycling plant in Eindhoven, in order to identify queueing problems that occur at the entrance gate, and within the WRP itself. Based on these simulations, we then aimed to find improvements that could increase the efficiency and traffic flow of this plant. To draw conclusions, we used raw data that was measured at the WRP in 2012.

First, we simulated the WRP using object-oriented programming in Python, after defining the model using a number of assumptions. Running the simulation 200 times gave us that, on average, the queue was too long, meaning longer than 3 cars, for 10800 out of a total 43200 seconds, which is 25% of the time. Each vehicle had an average waiting time of 800 seconds at the entrance gate. Due to the long waiting times, over 70% of a total average 137 vehicles queueing in front of the gate eventually left due to impatience. Although this can be true, looking at our model, this can also imply that the impatience parameter is set too high, or that the waiting times are not realistic.
As for the queues within the WRP, they are on average only 6 seconds, meaning that the traffic flow in the WRP is high. However, on average customers still spend over 800 seconds inside the WRP.

Taking these results into account, we can conclude that the WRP currently does not perform well for a decent number of vehicles. To solve this, we have found a number of solutions. One of them is reducing the service time within the WRP. Even though there are no queueing problems in the WRP itself, the total time spent inside is still high, causing long queues at the gate. Reducing this time, for instance by automating the waste dump process, would decrease total waiting times. In addition to this, there are a number of other possible improvements. Spreading customers throughout the day could decrease waiting times, by preventing high queue peaks, and expanding the WRP to have more parking spots will increase the total capacity, also minimising the queues.
In terms of cost, spreading customers is the cheapest option, while expanding the WRP would be the costliest option. Ultimately, a solution's cost would have to be compared to its efficiency to form a concrete conclusion.

A Python Code

A.1 Classes

```python
import numpy as np
from matplotlib import pyplot as plt
import copy
import scipy.stats as st
import heapq
import pandas as pd
from collections import deque
from numpy.ma.core import zeros

rng = np.random.default_rng()

# Creating path for input data
from google.colab import drive
drive.mount('/content/gdrive')
cd /content/gdrive/MyDrive

-------------------------------------------------------------------------------
class FES:

    def __init__(self):
        self.events = []

    def add(self, event):
        heapq.heappush(self.events, event)

    def next(self):
        return heapq.heappop(self.events)

    def isEmpty(self):
        return len(self.events) == 0

-------------------------------------------------------------------------------
class Event:

    ARRIVAL = 0
    STATIONMOVE = 1
    DEPARTURE = 2
    IMPATIENT = 3

    def __init__(self, typ, vehicle, time):
        self.type = typ
        self.vehicle = vehicle
        self.time = time

    def __lt__(self, other): # due to events possibly taking place at the same time, also sort them by type
        if self.time == other.time:
            if self.type == self.IMPATIENT or other.type == self.IMPATIENT:
                if self.type == self.IMPATIENT and other.type == self.IMPATIENT:
                    return self.vehicle.arrTime < other.vehicle.arrTime
                return self.type == self.IMPATIENT

            if self.type == self.DEPARTURE or other.type == self.DEPARTURE:
                if self.type == self.DEPARTURE and other.type == self.DEPARTURE:
                    return self.vehicle.arrTime < other.vehicle.arrTime
                return self.type == self.DEPARTURE

            if self.type == self.ARRIVAL or other.type == self.ARRIVAL:
```

```
                return self.type == self.ARRIVAL

            if self.vehicle.currentStation == 0 or other.vehicle.currentStation == 0:
                if self.vehicle.currentStation == 0 and other.vehicle.currentStation == 0:
                    if self.vehicle.priority == other.vehicle.priority:
                        return self.vehicle.arrTime < other.vehicle.arrTime
                    return self.vehicle.priority < other.vehicle.priority
                return other.vehicle.currentStation == 0

            if self.vehicle.currentStation.order == other.vehicle.currentStation.order:
                return self.vehicle.arrTime < other.vehicle.arrTime
            return self.vehicle.currentStation.order > other.vehicle.currentStation.order

        return self.time < other.time

-------------------------------------------------------------------------------
class Vehicle:

    def __init__(self, arrTime, stations, forgetCityPass, impatiencyTime, vehicleVars, typRNG):
        self.arrTime = arrTime
        self.forgetCityPass = forgetCityPass
        self.impatiencyTime = impatiencyTime
        self.gateWaitTime = 0
        self.stationArrTime = 0
        self.stationWaitTime = 0
        self.serviced = False
        self.stations = stations
        self.upcomingStations = copy.copy(stations)
        heapq.heapify(self.upcomingStations)
        self.currentStation = 0
        self.vehicleTypes = dict({})
        for i in range(len(vehicleVars)):
            self.vehicleTypes[list(vehicleVars.keys())[i]] = i
        self.type = 0
        self.typRNG = typRNG
        while self.typRNG > vehicleVars[list(vehicleVars.keys())[self.type]]:
            self.typRNG -= vehicleVars[list(vehicleVars.keys())[self.type]]
            self.type += 1
        if self.type == self.vehicleTypes["VAN"]:
            self.size = 2
        else:
            self.size = 1
        if self.type == self.vehicleTypes["PEDESTRIAN"] or self.type == self.vehicleTypes["BIKE"]:
            self.priority = 1
        else:
            self.priority = 2

    def setDepTime(self, depTime):
        self.depTime = depTime

    def nextStation(self):
        self.currentStation = heapq.heappop(self.upcomingStations)
        return self.currentStation

    def noStationsLeft(self):
        return len(self.upcomingStations) == 0

    def __lt__(self, other):
```

```
        if self.priority < other.priority:
            return True
        return self.arrTime <= other.arrTime

# ----------------------------------------------------------------------------
class GateQueue:

    def __init__(self):
        self.queue = []

    def add(self, vehicle):
        heapq.heappush(self.queue, vehicle)

    def next(self):
        return heapq.heappop(self.queue)

    def isEmpty(self):
        return len(self.queue) == 0

    def length(self):
        return len(self.queue)

    def remove(self, vehicle):
        self.queue.remove(vehicle)
        heapq.heapify(self.queue)

# ----------------------------------------------------------------------------
class Gate:

    def __init__(self, capacity, serviceTimeVars):
        self.capacity = capacity
        self.occupation = deque()
        self.queue = GateQueue()
        self.serviceTimeVars = serviceTimeVars

    def add(self, vehicle):
        if not self.isFull():
            self.occupation.append(vehicle)
        else:
            self.queue.add(vehicle)

    def queueToGate(self):
        vehicle = self.queue.next()
        self.occupation.append(vehicle)
        return vehicle

    def remove(self, vehicle):
        self.occupation.remove(vehicle)

    def isFull(self):
        return len(self.occupation) == self.capacity

    def fill(self):
        fill = 0
        for i in range(len(self.occupation)):
            fill += self.occupation[i].size
        return fill

    def relativeFill(self):
        return len(self.occupation) / self.capacity

    def returnQueueLen(self):
        return self.queue.length()

```

```python
    def genServiceTime(self):
        serviceTime = rng.gamma(self.serviceTimeVars[0], self.serviceTimeVars[1])
        return serviceTime

-------------------------------------------------------------------------------
class GarbageQueue:

    def __init__(self):
        self.queue = deque()

    def remove(self, vehicle):
        self.queue.remove(vehicle)

    def add(self, vehicle):
        self.queue.append(vehicle)

    def isEmpty(self):
        return len(self.queue) == 0

    def length(self):
        return len(self.queue)

-------------------------------------------------------------------------------
class Station:

    def __init__(self, name, order, capacity, serviceTimeVars):
        self.name = name
        self.order = order
        self.capacity = capacity
        self.occupation = deque()
        self.queue = GarbageQueue()
        self.serviceTimeVars = serviceTimeVars

    def __lt__(self, other):
        return self.order < other.order

    def fill(self):
        fill = 0
        for i in range(len(self.occupation)):
            fill += self.occupation[i].size
        return fill

    def fits(self, size):
        return (self.capacity - self.fill()) >= size

    def add(self, vehicle):
        if self.fits(vehicle.size) and self.queue.isEmpty():
            self.occupation.append(vehicle)
        else:
            self.queue.add(vehicle)

    def queueFits(self):
        return self.fits(self.queue.queue[0].size)

    def queueToStation(self):
        vehicle = self.queue.queue[0]
        self.queue.remove(vehicle)
        self.occupation.append(vehicle)
        return vehicle

    def remove(self, vehicle):
        self.occupation.remove(vehicle)

```

```
    def isFull(self):
        return self.fill() == self.capacity

    def relativeFill(self):
        return self.fill() / self.capacity

    def returnQueueLen(self):
        return self.queue.length()

    def genServiceTime(self):
        serviceTime = rng.gamma(self.serviceTimeVars[0], self.serviceTimeVars[1])
        return serviceTime

-------------------------------------------------------------------------------
class Simulation:

    def simulate(self, startTime, openTime, closeTime, stations, arrTimeVars, gateTimeVars, impatiencyVars, vehicleVars, gateEntrances = 1, forgetChance = 0.05, disableImpatiency = False):
        t = startTime
        fes = FES()

        stationLst = []
        for n in range(len(stations)):
            stationLst.append(Station(stations[n][0], stations[n][1], stations[n][2], stations[n][3]))
        gate = Gate(gateEntrances, gateTimeVars)

        self.createArrival(t, startTime, closeTime, arrTimeVars, impatiencyVars, stationLst, stations, forgetChance, vehicleVars, fes)

        queueLog = []
        impatientLog = []
        vehicleLog = []
        self.logQueues(queueLog, t, gate, stationLst)

        run = True
        while run:
            event = fes.next()
            t = event.time
            vehicle = event.vehicle
            if event.type == Event.ARRIVAL:
                gate.add(vehicle)
                if gate.queue.isEmpty():
                    # plan move to station
                    self.planGateMove(t, openTime, vehicle, gate, fes)
                elif not disableImpatiency:
                    # plan leaving due to impatiency
                    event = Event(Event.IMPATIENT, vehicle, vehicle.impatiencyTime)
                    fes.add(event)
                # plan next arrival
                self.createArrival(t, startTime, closeTime, arrTimeVars, impatiencyVars, stationLst, stations, forgetChance, vehicleVars, fes)

            elif event.type == Event.STATIONMOVE:
                oldStation = vehicle.currentStation
                if oldStation == 0: # the old station was the gate
                    allSpace = True # check if all stations have space
                    for n in range(len(stationLst)):
                        if not stationLst[n].fits(vehicle.size):
                            allSpace = False
                    if allSpace:
```

```
                        newStation = vehicle.nextStation()
                        # move to next station
                        newStation.add(vehicle)
                        if newStation.queue.isEmpty():
                            # instantly serviced at new station
                            self.planStationMove(t, vehicle, newStation, fes)
                        # manage situation at old location (gate)
                        gate.remove(vehicle)
                        if not gate.queue.isEmpty():
                            newVehicle = gate.queueToGate()
                            self.planGateMove(t, openTime, newVehicle, gate, fes)
                    else: # no space for vehicle so reinsert this event directly after the next time someone moves from a station
                        nextEventType = Event.ARRIVAL
                        nextEventStation = 0
                        i = -1
                        while not ((nextEventType == Event.DEPARTURE or nextEventType == Event.STATIONMOVE) and not nextEventStation == 0):
                            i += 1
                            nextEventType = fes.events[i].type
                            nextEventStation = fes.events[i].vehicle.currentStation
                        event.time = fes.events[i].time
                        vehicle.gateWaitTime += event.time - t
                        fes.add(event)
                else:
                    newStation = vehicle.nextStation()
                    # move to next station
                    newStation.add(vehicle)
                    if newStation.queue.isEmpty():
                        # instantly serviced at new station
                        self.planStationMove(t, vehicle, newStation, fes)
                    # manage situation at old location (station)
                    self.updateOldStation(t, vehicle, oldStation, fes)

            elif event.type == Event.DEPARTURE:
                oldStation = vehicle.currentStation
                if oldStation == 0:
                    # manage situation at old location (gate)
                    gate.remove(vehicle)
                    if not gate.queue.isEmpty():
                        newVehicle = gate.queueToGate()
                        self.planGateMove(t, openTime, newVehicle, gate, fes)
                else:
                    # manage situation at old location (station)
                    self.updateOldStation(t, vehicle, oldStation, fes)
                self.logVehicle(vehicleLog, vehicle)

            elif event.type == Event.IMPATIENT and not vehicle.serviced:
                gate.queue.remove(vehicle)
                self.logImpatient(impatientLog, t, vehicle, gate)

            self.logQueues(queueLog, t, gate, stationLst)
            if fes.isEmpty():
                run = False
        return queueLog, impatientLog, vehicleLog

    def createArrival(self, t, startTime, closeTime, arrTimeVars, impatiencyVars, stationLst, stations, forgetChance, vehicleVars, fes):
        halfHourOfService = np.floor(t / 30 / 60) - np.floor(startTime / 30 / 60) # find correct vehicle arrival density
        beta = 1 / (arrTimeVars[int(halfHourOfService)] / 30 / 60)
        arrRNG = rng.exponential(scale = beta)
```

```python
        emptyTimeslots = 0
        if arrRNG > 30 * 60 * (emptyTimeslots + 1): # with extremely sparse arrivals, a first arrival might take ages
            emptyTimeslots += 1                      # so it'll select the next density after half an hour
            beta = 1 / (arrTimeVars[int(halfHourOfService + emptyTimeslots)] / 30 / 60)
            arrRNG = emptyTimeslots * 30 * 60 + rng.exponential(scale=beta) / emptyTimeslots
        arrTime = t + arrRNG
        if arrTime <= closeTime: # as long as vehicles are allowed to arrive, create a new one
            impatiencyTime = arrTime + rng.gamma(impatiencyVars[0], impatiencyVars[1])
            typRNG = rng.uniform()
            vehicleStations = [] # generate stations to be visited and list cannot be empty
            while len(vehicleStations) == 0:
                for n in range(len(stationLst)):
                    if rng.uniform() <= stations[n][4]:
                        vehicleStations.append(stationLst[n])
            forgetCityPass = rng.uniform() <= forgetChance
            vehicle = Vehicle(arrTime, vehicleStations, forgetCityPass, impatiencyTime, vehicleVars, typRNG)
            event = Event(Event.ARRIVAL, vehicle, arrTime)
            fes.add(event)

    def planGateMove(self, t, openTime, vehicle, gate, fes):
        vehicle.gateWaitTime = t - vehicle.arrTime
        vehicle.serviced = True
        serviceTime = max(t, openTime) + gate.genServiceTime()
        if vehicle.forgetCityPass:
            vehicle.setDepTime(serviceTime)
            event = Event(Event.DEPARTURE, vehicle, serviceTime)
        else:
            vehicle.stationArrTime = serviceTime
            event = Event(Event.STATIONMOVE, vehicle, serviceTime)
        fes.add(event)

    def planStationMove(self, t, vehicle, station, fes):
        if vehicle.noStationsLeft():
            #plan departure
            serviceTime = t + station.genServiceTime()
            vehicle.setDepTime(serviceTime)
            event = Event(Event.DEPARTURE, vehicle, serviceTime)
            fes.add(event)
        else:
            #plan stationmove
            serviceTime = t + station.genServiceTime()
            vehicle.stationArrTime = serviceTime
            event = Event(Event.STATIONMOVE, vehicle, serviceTime)
            fes.add(event)

    def updateOldStation(self, t, vehicle, oldStation, fes):
        oldStation.remove(vehicle)
        if not oldStation.queue.isEmpty():
            if oldStation.queueFits():
                newVehicle = oldStation.queueToStation()
                newVehicle.stationWaitTime += (t - newVehicle.stationArrTime)
                self.planStationMove(t, newVehicle, oldStation, fes)

    def logQueues(self, log, t, gate, stationLst):
        logEntry = [t, len(gate.occupation), gate.fill(), gate.returnQueueLen()]
```

```
        for n in range(len(stationLst)):
            logEntry.append(len(stationLst[n].occupation))
            logEntry.append(stationLst[n].fill())
            logEntry.append(stationLst[n].returnQueueLen())
        log.append(logEntry)
        # print(logEntry)

    def logImpatient(self, log, t, vehicle, gate):
        logEntry = [t, vehicle.impatiencyTime - vehicle.arrTime, gate.
    returnQueueLen() + len(gate.occupation)]
        log.append(logEntry)

    def logVehicle(self, log, vehicle):
        logEntry = [vehicle.type, vehicle.arrTime, vehicle.depTime, vehicle.
    gateWaitTime, vehicle.stationWaitTime]
        if vehicle.forgetCityPass:
            subLogEntry = [0]
        else:
            subLogEntry = []
            for i in range(len(vehicle.stations)):
                subLogEntry.append(vehicle.stations[i].name)
        logEntry.append(subLogEntry)
        log.append(logEntry)

-------------------------------------------------------------------------------
```

A.2 Result functions

```
def getData(fileNames): # enter data into arrays and extract the name of the
    sheets
    allData = []
    sheets = []
    for i in range(len(fileNames)):
        allData.append(pd.read_excel(fileNames[i], sheet_name=None))
        sheets.append(list(allData[i].keys()))
    return allData, sheets

def getArrivalCounts(arrivalTimes): # filter arrivals in 30 minute intervals
    arrTimes = convertTime(arrivalTimes)
    arrVars = []
    for i in range(24 * 2):
        subArrTimes = arrTimes[arrTimes >= i * 30 * 60]
        subArrTimes = subArrTimes[subArrTimes < (i + 1) * 30 * 60]
        arrVars.append(len(subArrTimes))
    return arrVars

def getArrVars(allData, sheets): # get arrival counts per 30 minute interval as
     well as first arrival/opening/closing time
    arrivals = []
    openTime = 24 * 60 * 60
    for i in range(len(allData)):
        arrivalTimes = allData[i][sheets[i][1]]['Arrival'].to_numpy(na_value
    =-1).tolist()
        while len(arrivalTimes) > 0 and arrivalTimes[0] == -1:
            arrivalTimes.remove(-1)
        if len(arrivalTimes) > 0:
            arrivals.append(getArrivalCounts(arrivalTimes))
        openTimes = allData[i][sheets[i][0]]['Arrival'].to_numpy(na_value=-1).
    tolist()
        openTimes = convertTime(openTimes)
        openTimeIndex = 0
        while openTimes[openTimeIndex] == -1:
            openTimeIndex += 1
```

```
        openTime = min(openTime, openTimes[openTimeIndex])
    openTime = np.floor(2 * openTime / 60 / 60) / 2
    arrivals = np.mean(arrivals, axis=0).tolist()
    startTime = 0
    while arrivals[0] == 0:
        startTime += 0.5
        del arrivals[0]
    closeTime = 24
    while arrivals[-1] == 0:
        closeTime -= 0.5
        del arrivals[-1]
    return arrivals, startTime, openTime, closeTime

def getGateServiceVars(allData, sheets): # get gate service times and match them to a gamma distribution
    gateServiceTimes = []
    for i in range(len(allData)):
        vehicleIDs = allData[i][sheets[i][2]]['Car ID'].to_numpy(na_value=-1).tolist()
        gateArrivalTimes = allData[i][sheets[i][2]]['Entrance Time'].to_numpy(na_value=-1).tolist()
        gateArrivalTimes = convertTime(gateArrivalTimes)
        gateDepTimes = [24 * 60 * 60] * len(gateArrivalTimes)
        for j in range(len(sheets[i][3:])):
            keys = allData[i][sheets[i][3 + j]].columns
            stationVehicleIDs = allData[i][sheets[i][3 + j]][keys[0]].to_numpy(na_value=-1).tolist()
            stationArrivalTimes = allData[i][sheets[i][3 + j]][keys[1]].to_numpy(na_value=-1).tolist()
            stationArrivalTimes = convertTime(stationArrivalTimes)
            for k in range(len(stationVehicleIDs)):
                if stationVehicleIDs[k] != -1 and stationVehicleIDs[k] != 'Mbike' and stationVehicleIDs[k] != 'Foot':
                    index = 0
                    search = True
                    while stationVehicleIDs[k] in vehicleIDs[index + 1:] and search:
                        index = vehicleIDs.index(stationVehicleIDs[k], index + 1)
                        if gateArrivalTimes[index] < stationArrivalTimes[k]:
                            gateDepTimes[index] = min(stationArrivalTimes[k], gateDepTimes[index])
                            search = False
                        else:
                            gateDepTimes[index] = 0
                            search = False
        for j in range(len(gateDepTimes)):
            delta = gateDepTimes[j] - gateArrivalTimes[j]
            if delta < 2 * 60 and delta > 0:
                gateServiceTimes.append(delta)
    theta = np.var(gateServiceTimes) / np.mean(gateServiceTimes)
    k = np.mean(gateServiceTimes) / theta
    # randomSamples = rng.gamma(k, theta, 2000)
    # plt.hist(gateServiceTimes, weights = [1/len(gateServiceTimes)] * len(gateServiceTimes), bins = 100)
    # plt.hist(randomSamples, bins=100, density=True, alpha=0.5)
    # plt.show()
    return (k, theta)

def getStationServiceVars(allData, sheets): # get servicetimes for each station and match them to gamma distributions
    stations = sheets[0][3:]
    stationServiceVars = dict({})
    for j in range(len(stations)):
```

```
        arrTimes = []
        depTimes = []
        for i in range(len(allData)):
            keys = allData[i][sheets[i][3 + j]].columns.tolist()
            arrTimes.extend(allData[i][sheets[i][3 + j]][keys[1]].to_numpy(na_value=-1).tolist())
            depTimes.extend(allData[i][sheets[i][3 + j]][keys[2]].to_numpy(na_value=-1).tolist())
        arrTimes = convertTime(arrTimes)
        depTimes = convertTime(depTimes)
        deltaTimes = np.zeros(len(arrTimes))
        k = 0
        for i in range(len(arrTimes)):
            if arrTimes[i] != -1 and depTimes[i] != -1 and arrTimes[i] < depTimes[i] and depTimes[i] - arrTimes[i] < 30 * 60:
                deltaTimes[k] = depTimes[i] - arrTimes[i]
                k += 1
        deltaTimes = deltaTimes[:-deltaTimes.tolist().count(0)]
        theta = np.var(deltaTimes) / np.mean(deltaTimes)
        k = np.mean(deltaTimes) / theta
        stationServiceVars[sheets[0][3 + j]] = [k, theta]
    return stationServiceVars

def getGarbageAtStations(allData, sheets): # get the garbage collected at each station
    stations = sheets[0][3:]
    stationGarbage = [['T', 'Wc', 'Gr', 'Gs', 'E'], ['Dc', 'Dd', 'Gp', 'Ch'], ['F'], ['W', 'P', 'M'], ['Ca', 'Ma', 'Sl', 'A', 'R', 'Cl', 'B']]
    return stations, stationGarbage

def getGarbageTypeChance(allData, sheets): # get the chance each type of garbage is present
    mainSheetKeys = allData[0][sheets[0][0]].columns
    idIndex = mainSheetKeys[0]
    garbageTypes = mainSheetKeys[5:].tolist()
    garbageTypeChance = np.zeros(len(garbageTypes))
    nVehicles = 0
    for i in range(len(allData)):
        dataSheet = allData[i][sheets[i][0]]
        for j in range(len(dataSheet)):
            if not type(dataSheet[idIndex][j]) == float:
                nVehicles += 1
                for k in range(len(garbageTypes)):
                    if dataSheet[garbageTypes[k]][j] == 1 or dataSheet[garbageTypes[k]][j] == 0:
                        garbageTypeChance[k] += dataSheet[garbageTypes[k]][j]
    garbageTypeChance = garbageTypeChance / nVehicles
    garbageTypeChanceDict = dict({})
    for i in range(len(garbageTypes)):
        garbageTypeChanceDict[garbageTypes[i]] = garbageTypeChance[i]
    return garbageTypeChanceDict

def getForgetChance(allData, sheets): # finds 'pass' in the comments of the entrance sheet to see if anyone has forgotten their pass
    forgotten = 0
    nVehicles = 0
    for i in range(len(allData)):
        vehicles = allData[i][sheets[i][2]].to_numpy(na_value=-1)[:, 0].tolist()
        nVehicles += len(vehicles) - vehicles.count(-1)
        comments = allData[i][sheets[i][2]]['Comments']
        for j in range(len(comments)):
            if type(comments[j]) == str and 'pass' in comments[j]:
                forgotten += 1
```

```python
    return forgotten / nVehicles

def getImpatiencyVars(allData, sheets): # tries to find variables for the gamma distribution of the times people are willing to wait in queue
    deltaTimes = []                     # it does this by trying to match it in a crude machine learning type of manner
    queueTimes = []
    for i in range(len(allData)):
        keys = allData[i][sheets[i][1]].columns
        arrTimes = allData[i][sheets[i][1]][keys[2]].to_numpy(na_value=-1).tolist()
        depTimes = allData[i][sheets[i][1]][keys[3]].to_numpy(na_value=-1).tolist()
        vehicleIDs = allData[i][sheets[i][1]][keys[0]].to_numpy(na_value=-1).tolist()
        keys = allData[i][sheets[i][2]].columns
        gateArrTimes = allData[i][sheets[i][2]][keys[2]].to_numpy(na_value=-1).tolist()
        gateVehicleIDs = allData[i][sheets[i][2]][keys[0]].to_numpy(na_value=-1).tolist()
        arrTimes = convertTime(arrTimes)
        depTimes = convertTime(depTimes)
        gateArrTimes = convertTime(gateArrTimes)
        for j in range(len(depTimes)):
            if depTimes[j] != -1 and arrTimes[j] != -1:
                deltaTimes.append(depTimes[j] - arrTimes[j])
            elif arrTimes[j] != -1 and vehicleIDs.count(vehicleIDs[j]) == 1 and gateVehicleIDs.count(vehicleIDs[j]) == 1:
                index = gateVehicleIDs.index(vehicleIDs[j])
                if gateArrTimes[index] != -1 and gateArrTimes[index] - arrTimes[j] > 0:
                    queueTimes.append(gateArrTimes[index] - arrTimes[j])
    attempts = 250
    lastImprovement = 0
    runs = 1
    maxRuns = 10000
    startScale = 5
    repeats = 10
    totalVehicles = len(queueTimes)
    totalImpatient = len(deltaTimes)
    oldTheta = np.var(deltaTimes) / np.mean(deltaTimes)
    oldK = np.mean(deltaTimes) / oldTheta
    oldErr = 0
    for i in range(repeats):
        randomSamples = rng.gamma(oldK, oldTheta, totalVehicles)
        for j in range(totalVehicles):
            if randomSamples[j] < queueTimes[j]:
                oldErr += 1
    oldErr = oldErr / repeats - totalImpatient
    while lastImprovement < attempts and runs < maxRuns:
        theta = max(oldTheta + rng.normal(scale = startScale/math.sqrt(runs)), 0)
        k = max(oldK + rng.normal(scale = startScale/math.sqrt(runs)), 0)
        err = 0
        for i in range(repeats):
            randomSamples = rng.gamma(k, theta, totalVehicles)
            for j in range(totalVehicles):
                if randomSamples[j] < queueTimes[j]:
                    err += 1
        err = err / repeats - totalImpatient
        if abs(err) < abs(oldErr):
            oldErr = err
            oldTheta = theta
            oldK = k
```

```
            lastImprovement = 0
        else:
            lastImprovement += 1
        runs += 1
    plt.hist(queueTimes)
    plt.show()
    # plt.hist(rng.gamma(oldK, oldTheta, 1000))
    # plt.show()
    # print(oldErr)
    return [oldK, oldTheta]

def getVehicleVars(allData, sheets): # get chances each type of vehicle shows
    up
    nVehicles = 0
    nVans = 0
    nFoot = 0
    nBike = 0
    for i in range(len(allData)):
        keys = allData[i][sheets[i][2]].columns
        vehicleIDs = allData[i][sheets[i][2]][keys[0]].to_numpy(na_value=-1).
    tolist()
        keys = allData[i][sheets[i][0]].columns
        sizes = allData[i][sheets[i][0]][keys[1]].to_numpy(na_value=-1).tolist
    ()
        nVehicles += len(vehicleIDs) - vehicleIDs.count(-1)
        nVans += sizes.count('B')
        for j in range(len(vehicleIDs)):
            if type(vehicleIDs[j]) == str:
                if "FOT" in vehicleIDs[j] or "Foot" in vehicleIDs[j]:
                    nFoot += 1
                elif "BIK" in vehicleIDs[j] or "Bike" in vehicleIDs[j] or "
    Mbike" in vehicleIDs[j]:
                    nBike += 1
    nCars = nVehicles - nVans - nFoot - nBike
    varDict = dict({})
    varDict['VAN'] = nVans / nVehicles
    varDict['CAR'] = nCars / nVehicles
    varDict['PEDESTRIAN'] = nFoot / nVehicles
    varDict['BIKE'] = nBike / nVehicles
    return varDict

def convertTime(times): # convert times from the excel sheet into seconds since
     start of day
    convertedTime = np.zeros(len(times))
    for i in range(len(times)):
        if times[i] == -1:
            convertedTime[i] = -1
        elif type(times[i]) == float or type(times[i]) == int:
            timeStr = str(int(times[i]))
            convertedTime[i] = (int(timeStr[0:-4]) * 60 + int(timeStr[-4:-2]))
    * 60 + int(timeStr[-2:])
        elif type(times[i]) == datetime.time:
            convertedTime[i] = (times[i].hour * 60 + times[i].minute) * 60 +
    times[i].second
        elif type(times[i]) == str:
            if times[i].count(':') == 2:
                lIndex = times[i].find(':')
                rIndex = times[i].rfind(':')
                convertedTime[i] = (int(times[i][lIndex - 2:lIndex]) * 60 + int
    (times[i][lIndex + 1:rIndex])) * 60 + int(times[i][rIndex + 1:rIndex + 3])
            else:
                convertedTime[i] = -1
        else:
            print(times[i])
```

```
            print(type(times[i]))
    return convertedTime
```

A.3 Executables

```
def stepPlot(t, y, y2, title, yunit): # plot queue and occupancy over time
    plt.step(t, y + y2, label = 'queue')
    plt.step(t, y, label = 'occupancy')
    start = int(np.floor(min(t)))
    stop = int(np.ceil(max(t)))
    xValues = range(start, stop + 1, 1)
    plt.xticks(ticks = xValues, labels = xValues)
    yValues = range(0, int(max(y + y2) + 1), 1)
    plt.yticks(ticks = yValues, labels = yValues)
    plt.legend()
    plt.grid(True)
    plt.title(title)
    plt.xlabel('time (hour)')
    plt.ylabel('occupancy (' + yunit + ')')
    plt.show()

def histPlot(x, title, xlabel, ylabel, bins = 30, density = True, weights =
    None): # plot histogram
    plt.hist(x, bins = bins, density = density, log = False, weights = weights)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.show()


# get all variables
fileNames = ['04-09-12RawData.xlsx', '06-09-12RawData.xlsx', '07-09-12RawData.
    xlsx']
allData, sheets = getData(fileNames)
arrivals, startTime, openTime, closeTime = getArrVars(allData, sheets)
garbageTypeChanceDict = getGarbageTypeChance(allData, sheets)
stations, stationGarbage = getGarbageAtStations(allData, sheets)
gateServiceVars = getGateServiceVars(allData, sheets)
vehicleVars = getVehicleVars(allData, sheets)
forgetChance = getForgetChance(allData, sheets)
stationServiceVars = getStationServiceVars(allData, sheets)
impatiencyVars = getImpatiencyVars(allData, sheets)

#setup simulation
sim = Simulation()

# set variables
gateTimeVars = gateServiceVars
startTime = startTime * 60 * 60
openTime = openTime * 60 * 60
closeTime = closeTime * 60 * 60
arrTimeVars = arrivals
forgetChance = forgetChance
impatiencyVars = impatiencyVars
vehicleVars = vehicleVars

# set stations
stationLst = []
stationOrder = ['FWPM1', 'FWPM2', 'DcDdGpCh', 'GrWcTEGs', 'Rest + Exit']
stationCapacity = [2, 2, 4, 3, 2]
for i in range(len(stationOrder)):
    stationIndex = stations.index(stationOrder[i])
    stationChance = 1
```

```
    for j in range(len(stationGarbage[stationIndex])):
        stationChance = stationChance * (1 - garbageTypeChanceDict[stationGarbage[stationIndex][j]])
    stationChance = 1 - stationChance
    stationLst.append([stationOrder[i], i + 1, stationCapacity[i], stationServiceVars[stationOrder[i]], stationChance])

# set number of gate entrances (they share the same queue)
gateEntrances = 1

# set up the repeated simulation for results
nRuns = 200
gateQueueLimit = 3
meanQueueLength = np.zeros((1 + len(stations), nRuns))
meanTimeQueueTooLong = np.zeros(nRuns)
nVehicles = np.zeros(nRuns)
impatientVehicles = np.zeros(nRuns)
impatientTimes = []
gateWaitingTimes = []
stationWaitingTimes = []
serviceTimes = []
for i in range(nRuns):
    queueLog, impatientLog, vehicleLog = sim.simulate(startTime, openTime, closeTime, stationLst, arrTimeVars, gateTimeVars, impatiencyVars, vehicleVars, gateEntrances = gateEntrances, forgetChance = forgetChance, disableImpatiency = False)
    # process data from simulation
    queueLog = np.array(queueLog)
    impatientLog = np.array(impatientLog)
    vehicleLog = np.array(vehicleLog)
    for j in range(len(queueLog) - 1):
        meanTimeQueueTooLong[i] += ((queueLog[j, 1] + queueLog[j, 3]) > gateQueueLimit + gateEntrances) * (queueLog[j + 1, 0] - queueLog[j, 0])
        for k in range(len(stations) + 1):
            meanQueueLength[k][i] += queueLog[j, 3 + k * 3] * (queueLog[j + 1, 0] - queueLog[j, 0]) / (queueLog[-1][0] - queueLog[0][0])
    impatientVehicles[i] = len(impatientLog)
    for j in range(len(impatientLog)):
        impatientTimes.append(impatientLog[j][1])
    nVehicles[i] = len(vehicleLog) + impatientVehicles[i]
    for j in range(len(vehicleLog)):
        gateWaitingTimes.append(vehicleLog[j][3])
        stationWaitingTimes.append(vehicleLog[j][4])
        serviceTimes.append(vehicleLog[j][2] - vehicleLog[j][3] - vehicleLog[j][1])

# generate outputs
histPlot(gateWaitingTimes, 'Gate waiting times', 'waiting time (s)', 'occurrences', density = False, weights = [1/ nRuns] * len(gateWaitingTimes))
output = 'In ' + str(nRuns) + ' runs the mean time (s) when the gate queue is too long is ' + str(np.mean(meanTimeQueueTooLong)) + ' ± ' + str(np.std(meanTimeQueueTooLong)) + '.\n'
output += 'From an average of ' + str(np.mean(nVehicles)) + ' ± ' + str(np.std(nVehicles)) + ' vehicles, an average of ' + str(np.mean(impatientVehicles)) + ' ± ' + str(np.std(impatientVehicles)) + ' left the queue,\n'
output += 'these had an average waiting time (s) of ' + str(np.mean(impatientTimes)) + ' ± ' + str(np.std(impatientTimes)) + '. Vehicles usually spent ' + str(np.mean(gateWaitingTimes)) + ' ± ' + str(np.std(gateWaitingTimes)) + ' seconds waiting at the gate\n'
output += 'and ' + str(np.mean(stationWaitingTimes)) + ' ± ' + str(np.std(stationWaitingTimes)) + ' seconds waiting at stations. They also spent an average of ' + str(np.mean(serviceTimes)) + ' ± ' + str(np.std(serviceTimes)) + ' seconds inside the WRP.'
print(output)
```

```

stepPlot(queueLog[:, 0] / 60 / 60, queueLog[:, 1], queueLog[:, 3], 'Gate nr of
    cars', 'cars')
for n in range(len(stations)):
    stepPlot(queueLog[:, 0] / 60 / 60, queueLog[:, n * 3 + 4], queueLog[:, n *
    3 + 6], stationOrder[n] + ' nr of cars', 'cars')
```