A Heuristic for the Situational Application of a Destination Dispatch Elevator System

Ryan Do

April 2020

Contents

Introduction

In the realm of traditional elevator control algorithms, Highest Unanswered Floor First (HUFF) is a widely implemented one that has been shown to demonstrate a competitive performance even when compared to more sophisticated alternatives [3]. In HUFF, elevators start at either extreme of the chute and proceed in a single direction until all requests are answered.

An increasingly popular elevator control technique known as destination dispatch (DD) moves the destination buttons from within the car to the waiting area, making buttons absent inside the cars. In this set-up, information on a patron's intended destination is available to the system as soon as they register on a panel on each floor (left). DD has the advantage of knowing each patron's intended route so that cars can effectively prioritize pickup and drop-off routes according to demand. This also eliminates the scenario of cars being at capacity upon stopping to pick up passengers. Additionally, since each DD elevator only takes a single destination at a time, there are less stops en route. However, it is unclear in what scenarios a destination dispatch system is advantageous over the traditional HUFF algorithm. The objective of this study is to determine a general heuristic for selecting between destination dispatch or HUFF based on the properties of some building in question. This report will firstly dive deep into the simulation modelling approach of the elevator systems. The experimental design will then be covered, ending with the results of the study and conclusions.

Problem Description

In specific, the problem to be solved is to determine for what set of building parameters values (characterized by number of floors and patrons per floor) destination dispatch is to be recommended over traditional HUFF, given a typical office building arrival process. DD is recommended for some set of building dimensions if less elevator cars are required for the system performance to satisfy some criteria in simulation. The experimental design section elaborates on more of the details.

Model Description

Simulation Logic

When modelling HUFF and destination dispatch, certain aspects were controlled to not obscure the root source of potential performance differences. Consistent behavior between HUFF and DD include idling on the spot, idle cars being prioritized, and to prioritize floors closest to terminal ends first. In addition, the NSPP arrival process and physical properties are also made to be consistent.

Traditional Highest Unanswered Floor First (HUFF) Elevator

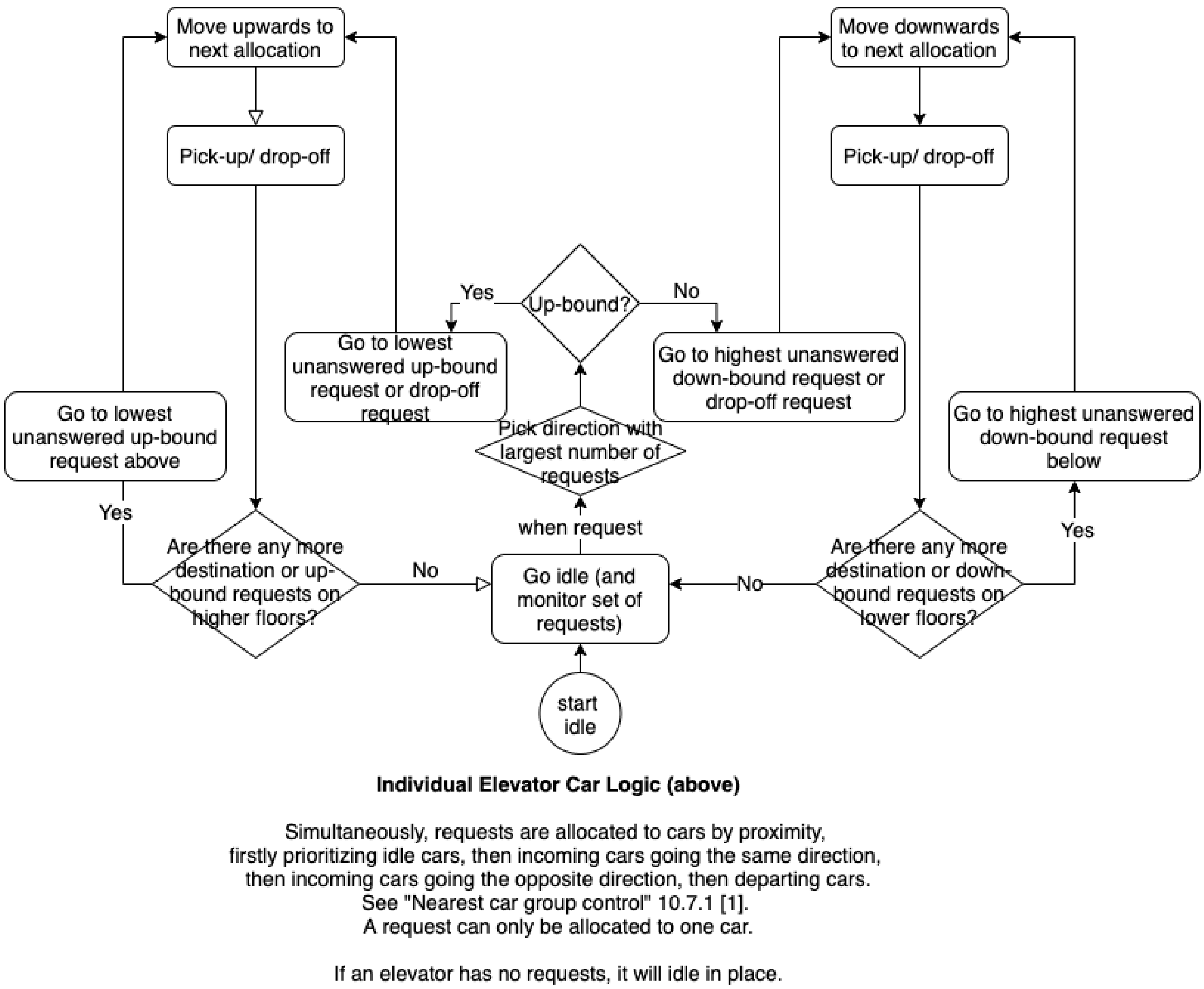

Figure 1. A complete description of the implemented HUFF logic as a flowchart.

In a nutshell, the HUFF system is implemented in simulation as two decision-making agent classes: (1) a controller dispatches pick-up requests to elevator cars based on prioritization rules, and (2) elevator cars complete the pool of requests allocated to each one in an intelligent order (Figure 1 for more detail). Some additional details and implications of the above logic include the following:

1. While a car is in transit to the next destination, if any new requests to pick-up from floors along the way are allocated to it, it will *not* be interrupted.
2. If a car is or becomes full when attempting to pick up from its current floor, the left-over waiting passengers re-submit their request(s).
3. Cars tend to complete all requests in one direction before switching directions.

Destination Dispatch (DD)

Note that there is little publicly available information on the exact detailed algorithms implemented in real destination dispatch systems. The algorithm detailed in Figure 2 is an informed assumption based on observation and a conceptual knowledge on its expected operation. Similarly to HUFF, two intelligent agents are implemented: elevator car and controller. The elevator car logic for DD is the same as HUFF. Cars look into their pool of requests, pick the direction with the largest number and start from the highest or lowest floor. The controller maintains a pool of unallocated tasks and cars allocate themselves tasks when they can take them. In addition, a couple custom data structures were designed to implemented DD efficiently (Figure 3). When an elevator is idle and ready to take a new task, it scans a central *source-destination matrix* and picks the column (destination) with the greatest sum.

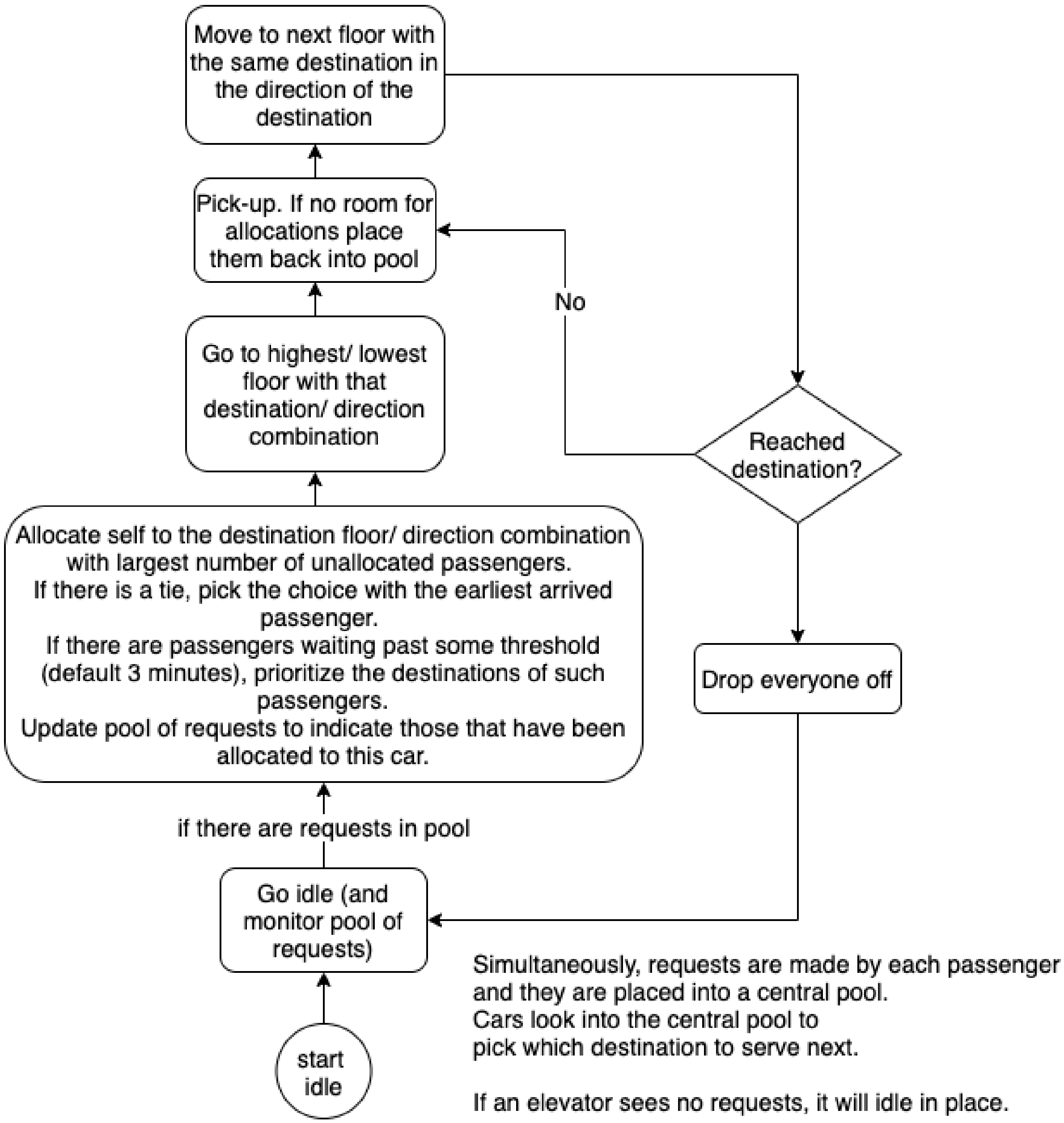

Figure 2. *A complete description of the implemented destination dispatch system.*

Those requests are then placed in its own *request pool*. The *source-destination queue-matrix* serves as a set of priority queues to rank destination floors by urgency and identify those with patrons waiting past some urgency threshold (3 minutes) or to identify highest time waiting patrons to act as a tie breaker for the greatest sum.

0	2	0
0	0	1
0	0	0

0	P1, P2	P3
0	0	P4, P5
0	0	0

0	1
0	1
0	0

Figure 3. Left: Source-destination matrix - number of unallocated requests by source (rows) and destination (column). Middle: Source-destination queue-matrix – priority queue of waiting patrons for each (source, destination) pair. Right: Request pool – pick-up requests for each source floor (rows) and direction (columns). One per elevator car.

Additional details and implications of the destination dispatch logic:

1. Cars should never allocate themselves to more passengers than expected. Therefore, there could be multiple cars going to the same ultimate destination.
2. On the other hand, if there is room for any unallocated passengers going to same destination, pick them up too. For example, the car in Figure 3 will pick up 2 passengers on floor 2 going to floor 3 despite only being allocated to 1.

System Properties

Input	Quantity	Units
Elevator car top speed	3.0	ms^{-1}
Elevator car acceleration/ deceleration	4.5	ms^{-2}
Inter-floor distance	4.5	m
Door open-close time	2.5	s
Time for one passenger to enter/exit car	1.0	s
Car capacity	20	passengers
Number of elevator cars	Experimental variable	#
Number of floors	Experimental variable	#
Occupants per floor	Experimental variable	#

Non-stationary arrival processes

The time-varying arrival rates for upwards traffic, downwards traffic, and cross-floor traffic are modeled by three non-stationary Poisson processes, generated using the thinning method. The arrival processes are modeled after the daily patterns of an office building, inspired by Barney section 4.4, 6.5 [1]:

	8AM – 9AM	9AM – 12:00PM	12PM – 1PM	1PM– 2PM	2PM – 6PM	6PM – 8AM
Per 5 minute % of building population	Upwards traffic (From ground floor to any floor)					
	8	0.125	2.75	5.5	0.125	0
	Downwards traffic (From any floor to ground floor)					
	0.125	5.5	2.75	0.125	8	0
	Intra-building traffic (From any floor to any floor)					
	1.0	1.0	1.0	1.0	1.0	0

Common Random Numbers

CRNs are applied to all replications to reduce the variance of the difference between scenarios through increasing covariance, thus increasing the probability of correct selection in the elevator fleet sizing procedure that follows. In situations where compared scenarios have differing performing servers, a single RNG stream may become out of sync. However in the case here, a single stream is sufficient since the shared inter-arrival times and source/ destination floors are the only random variables.

Model Assumptions

See A.1 for a list of assumptions made for the simulation models.

Experimental Design

The required number of elevator cars (fleet size) for DD is to be evaluated and compared to HUFF for each pair in {3, 5, 10, 15, 20, 25, 30} floors X {25, 50, 100, 200, 300} occupants per floor. Each pair in the cartesian product constitutes one scenario. For each scenario, the fixed budget strategy is used with a single replication to minimize bias. A fixed budget of a simulated 2 days is selected as it covers two full work-day cycles and the run-time is sufficiently small such that all scenarios can be run in a timely manner. Fixed-precision offers little benefit since run-time is costly. As a result, reaching a target precision is a luxury, and minimizing bias via 1 replication is priority. Local minimization of the MSER is used to determine the warmup period:

$$MSER(d) = \frac{1}{(m-d)^2}\sum_{i=d+1}^{m}\left(Y_i - \bar{\bar{Y}}(m,d)\right)^2 = \sum_{i=d+1}^{m} Y_i^2 - \frac{1}{m-d}\sum_{i=d+1}^{m}\left(Y_i - \bar{\bar{Y}}\right)^2$$

Fleet size requirements for each of DD and HUFF are evaluated on two different criteria, each describing different passenger needs. The first criteria is for the system to achieve < 50s passenger waiting time to enter the elevator with a 95% probability at 95% confidence. Along the same lines, the second criteria is for the system to achieve a total *journey time* of less than some threshold with 95% probability at 95% confidence, with this threshold defined to scale linearly with the number of floors in this case (since journey time includes movement between floors and so this threshold must be adaptive).

For some system, this threshold is defined to be $3\frac{h_{building}}{v_{max}} + 50\,seconds$. The CI of probability $\boldsymbol{\hat{F}}$ of a performance measure Y (wait time or journey time) not surpassing some threshold is: $\boldsymbol{\hat{F}} = \frac{1}{n}\sum_{i=1}^{n} I(Y_i \leq \text{threshold})$; $\boldsymbol{S^2} = \left(\frac{n}{n-1}\right)\hat{F}\left(1-\hat{F}\right)$; $\boldsymbol{CI} = \hat{F} \pm z_{1-\frac{\alpha}{2}}\frac{S}{\sqrt{n}}$

Results

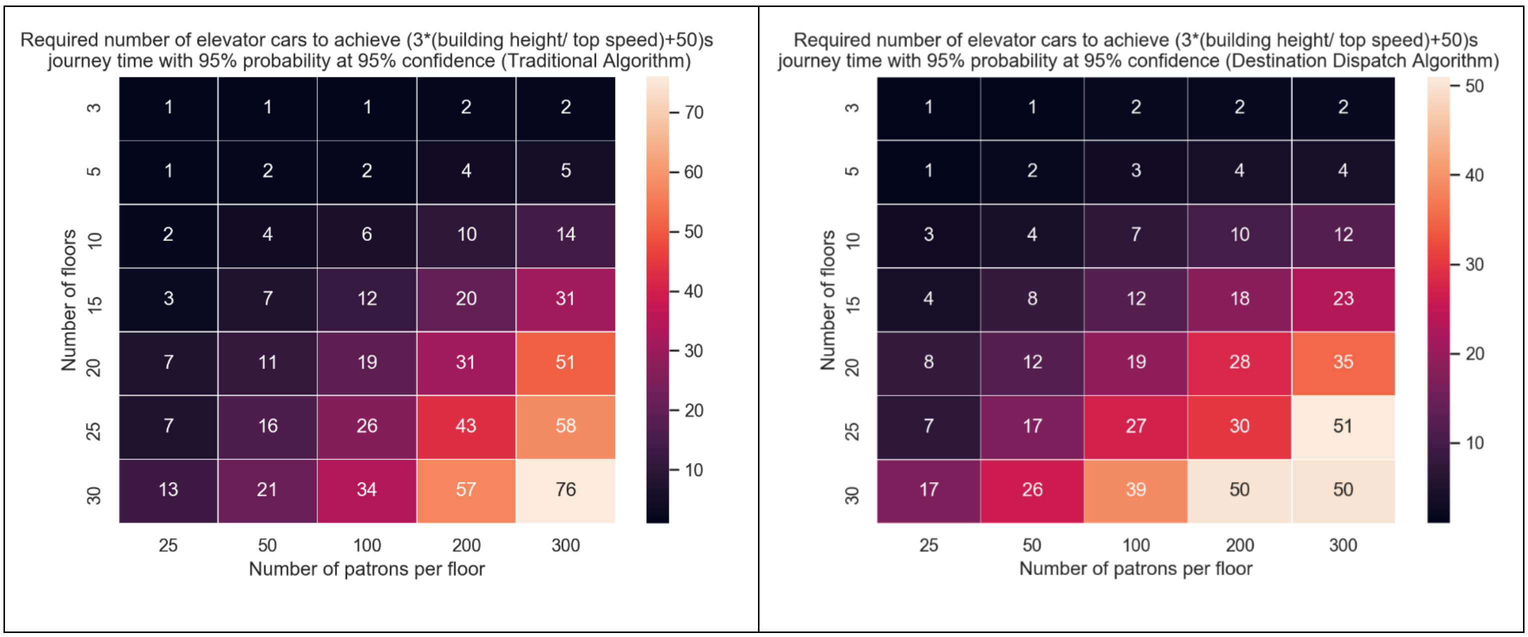

Figure 4. Sizing requirements determined by journey time; HUFF (left), DD (right).

The fleet sizing values shown in Figures 4 and 5 are determined through a binary search where an initial space of [0, 100] cars is recursively halved and replication run until the minimum number of cars to achieve the respective criteria is found. It is clear that with increasing floor count and floor population the required fleet size increases in all cases. Some key insights are more evident when visualizing the relative improvement of DD over HUFF (Figure 6). Negative values indicate a smaller quoted fleet sizing for DD,

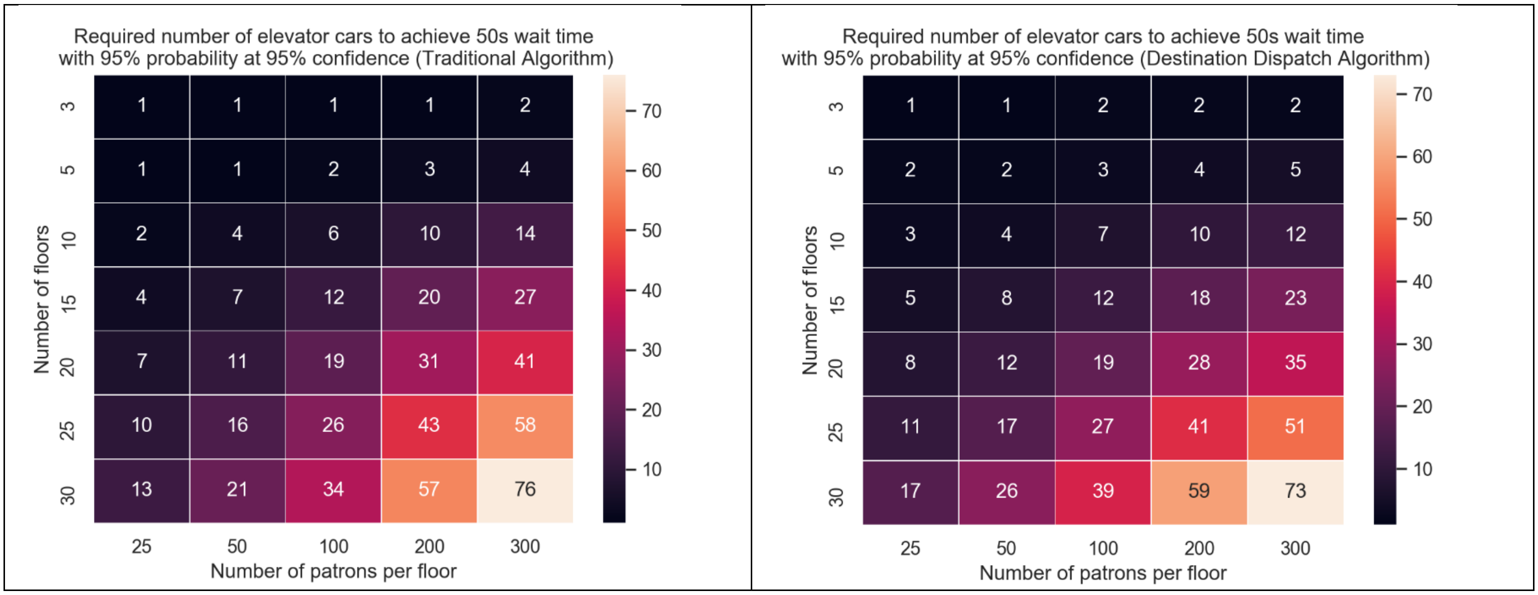

Figure 5. Sizing requirements determine by patron wait time; HUFF (left), DD (right).

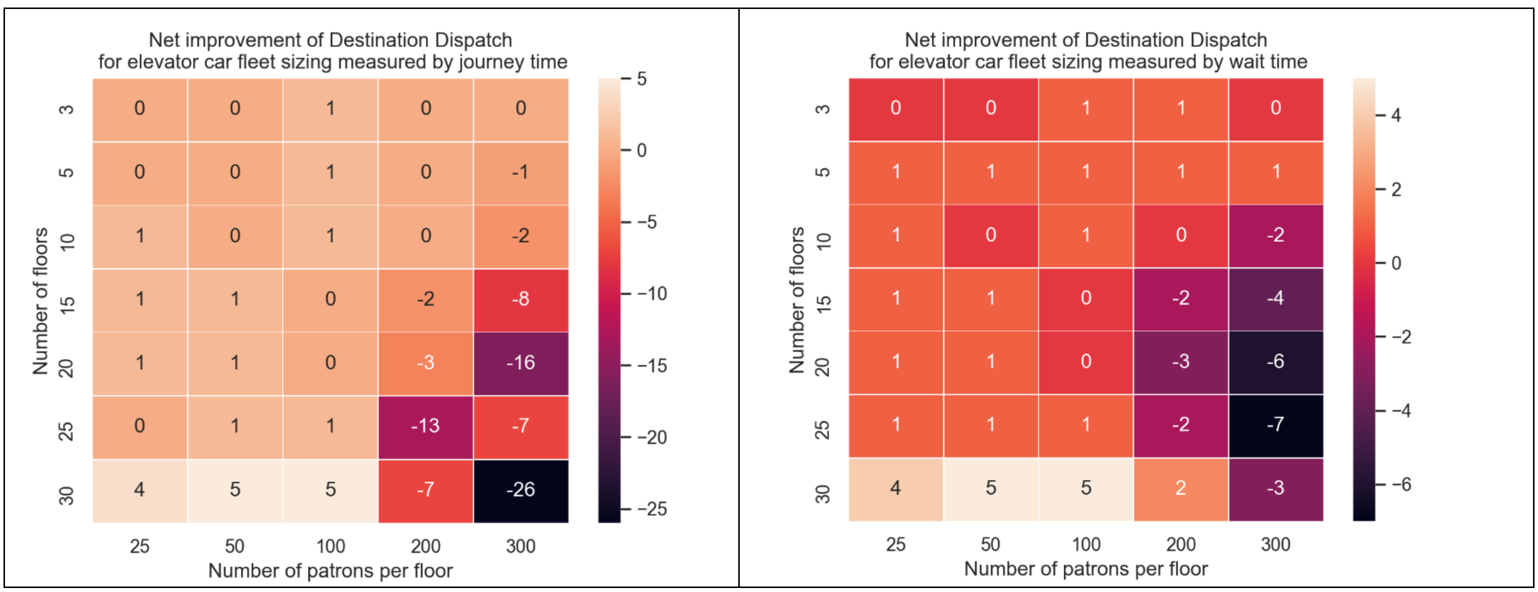

Figure 6. Delta in fleet sizing requirements from HUFF to DD for each criterion.

which represent the space of building properties where DD is recommended over HUFF. Figure 6 shows that whether evaluating by journey time or wait time, buildings with a large number of floors (10+) *and* a large occupancy per floor (150-200+) lend well to the implementation of a destination dispatch system. Using wait time as a performance measure tends to be more forgiving to the HUFF algorithm since destination dispatch trades off low wait time for a more significantly lower travel time. The simulated benefit of DD is very substantial when sizing by journey time, demonstrating up to a 26 elevator car reduction at higher floor populations. In cases where there is a low floor population and higher floor count, HUFF actually performs better regardless of criteria. At very high floor occupancy (300+), DD is preferred regardless of floor count. As such, high floor occupancy is the principal determinant. To make this effect intuitive, at the high end of floor occupancy, DD cars can run at capacity (since there are more patrons going to each floor) while HUFF cars are also at capacity. However, DD cars only take on one destination at a time therefore journey time and as a consequence waiting times are lower.

As a secondary take-away, periods of time with high upwards traffic from a single floor (i.e. 8-9 AM in Figure 7) show *even greater marginal performance increase* in DD since DD cars only take destinations one a time while a single HUFF car can have requests to all floors. In Figure 7, high down-peak traffic (5-6PM) show similar queue sizes between DD and HUFF (controlling all other variables) while in high up-peak traffic (8-9AM) there is a 6-fold decrease in max queue size when opting for destination dispatch. In conclusion, building structures with a high occupancy per floor

(200+), moderate to high floor count (10+), and high upwards traffic are especially well suited for destination dispatch. Luxury cruise ships, for example, fulfill all conditions.

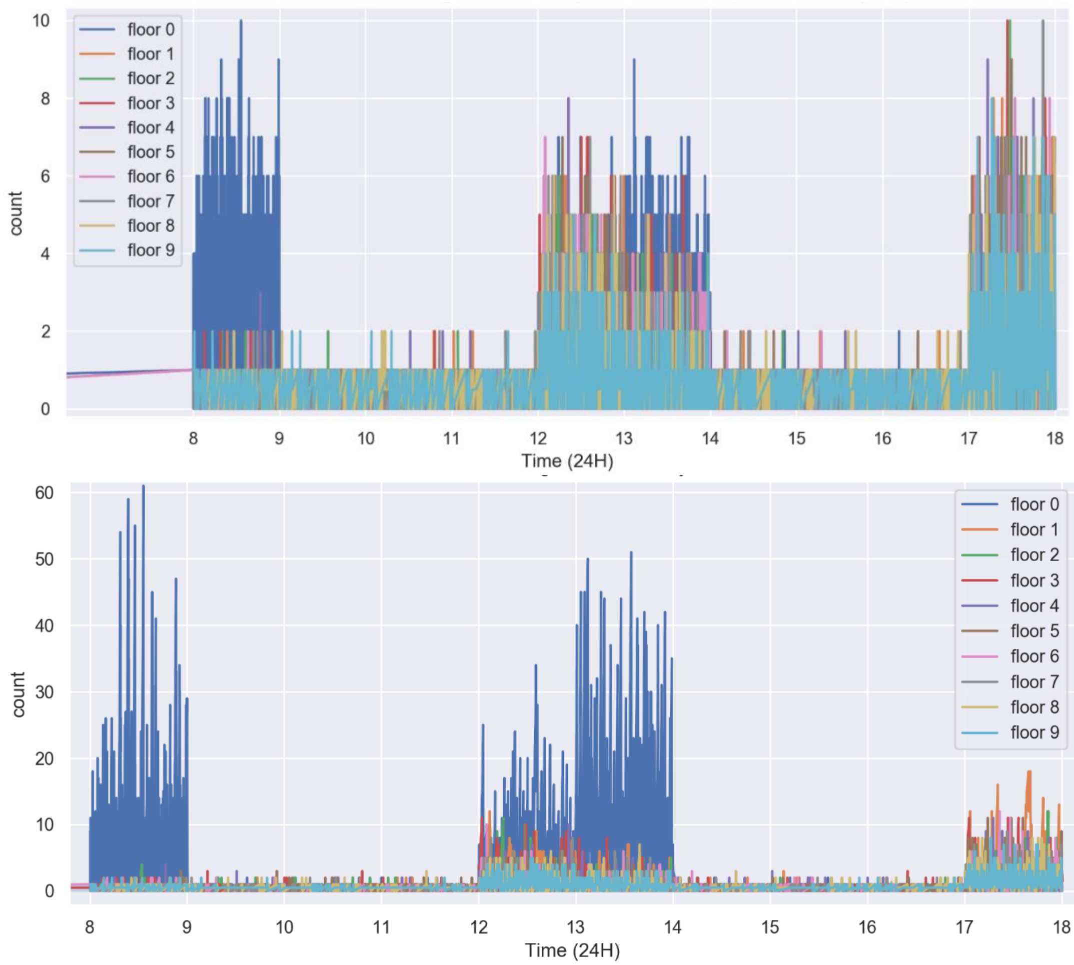

Figure 7. Queue sizes by floor over a single work day. DD (top), HUFF (bottom). 10 floors, 300 occupancy per floor, 12 cars for both.

References

[1] Barney, G. C., & Al-Sharif, L. (2016). *Elevator traffic handbook: theory and practice.*

[2] B. Nelson. (2013) Foundations and Methods of Stochastic Simulation:A First Course.

[3] R. Crites, A. Barto. (1996) *Improving Elevator Performance Using Reinforcement Learning.* NeurIPS 1996, Conference on Neural Information Processing Systems.

Appendix

A1. Model Assumptions

Passenger/ Building Assumptions

1. Equal demand to and from all non-ground floors for inter-floor traffic (uniformly distributed).
2. For destination dispatch, passengers will enter the car bound for the specific destination that they called.
3. For destination dispatch, each passenger in a group will press their floor individually.
4. For the traditional elevator, passengers will enter the first car going in their specified direction.
5. The building population per floor and arrival process are modelled after typical office buildings.
6. Building patrons do not ever use the stairs.

Elevator Logic Asssumptions

7. Elevator cars are not optimized to park idle in specific locations. They idle in place.

8. Elevator cars park idle with doors open. (whenever elevator cars move, it is guaranteed that they close the doors, move, then open the doors)

Elevator Dynamics Assumptions

9. Elevator cars have infinite jerk; they reach maximum acceleration instantaneously.
10. Elevator cars accelerate at the same magnitude as deceleration.
11. Elevator car acceleration is same up-bound and down-bound.
12. Elevator kinematic properties are based on Table 5.2 with 4.5m inter-floor distance and 60m travel [1]
13. If there is insufficient distance from source to destination to acceleration and deceleration to/from top speed, the car accelerates for half the distance then decelerates for the other half.

A2. Model Verification Checklist

1. Traditional elevator logic is correct:

   a. Requests are allocated to cars appropriately based on priority rules

   b. Full cars will reject passengers, and passenger requests are reallocated

   c. Cars continue in one direction whenever possible before switching directions

   d. If a car has both picks and drops on a floor, it will do both starting with drop

```
Executed event: <elevator_car_traditional.ElevatorCarTraditional.MoveEvent object at 0x121469950> Current time: 484.1796721485067, Post-event
 state below
Passengers waiting (source floor, destination floor) [(0, 2), (0, 1), (0, 2), (0, 3), (0, 2), (0, 2), (0, 3), (0, 4), (0, 3), (0, 4), (0, 3),
 (0, 2), (0, 2), (0, 1), (0, 3), (0, 4), (0, 4), (0, 1), (0, 1), (0, 1), (0, 2), (0, 4), (0, 1), (0, 4), (0, 2), (0, 4), (0, 3), (0, 1), (0, 3),
 (0, 2), (0, 3), (0, 1), (0, 1), (0, 4), (0, 1), (0, 1), (0, 4), (0, 3), (0, 4), (0, 4), (0, 2), (0, 1), (0, 4), (0, 4), (0, 1), (0, 2), (0, 1),
 (0, 2), (0, 4), (0, 2), (0, 2), (0, 2), (0, 3), (0, 1), (0, 4), (0, 3), (3, 4), (3, 1), (3, 2)]
Car 1 - onboard:[0, 0, 6, 8, 2] floor: 1 next floor: 2 status: 1 direction: 1
[array([[0., 1.],
       [0., 0.],
       [0., 0.],
       [1., 1.],
       [0., 0.]])]
```

2. Destination dispatch logic is correct:

   a. Car picks the destination/ direction combination with largest number of requests, uses earliest arrival as tie-breaker

```
Executed event: <traditional_elevator.ElevatorCarTraditional.DropoffEndEvent object
Source destination queue matrix count
 [[0 2 3 1 1]
 [0 0 0 1 0]
 [0 0 0 0 0]
 [0 0 1 0 0]
 [0 0 0 0 0]]
Source destination matrix
 [[0 0 0 1 1]
 [0 0 0 1 0]
 [0 0 0 0 0]
 [0 0 1 0 0]
 [0 0 0 0 0]]
Car 1 - onboard:[0, 0, 0, 0, 0] floor: 1 next floor: None status: 3 direction: 1 fi
Car 2 - onboard:[0, 0, 0, 0, 0] floor: 2 next floor: 0 status: 1 direction: 0 final
[array([[0., 2.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]]),
 array([[0., 3.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])]
import sys; print('Python %s on %s' % (sys.version, sys.platform))
Python 3.7.4 (default, Jul  9 2019, 18:13:23)
In[2]: self.source_destination_queue_matrix[0,1]
Out[2]: <custom_library.FIFOQueuePlus at 0x11f8aff90>
In[3]: self.source_destination_queue_matrix[0,1].ThisQueue[0].CreateTime
Out[3]: 492.64814183551835
In[4]: self.source_destination_queue_matrix[0,3].ThisQueue[0].CreateTime
Out[4]: 493.3753999294718
In[5]: self.source_destination_queue_matrix[1,3].ThisQueue[0].CreateTime
Out[5]: 492.96127689155094
```

Here, floor 1 and 3 (columns 1 and 3 with zero index in the source destination queue matrix) both have 2 patrons waiting to go there. However, floor 1 has the earlier arriving patron, coming in at t=492.64814. As a result, car 1 is allocated to floor 1 as the destination, and goes to floor 0 to pick up the 2 passengers.

b. Only as many passengers a car is able to handle is allocated to each car:

```
Executed event: <replication.ReplicationDestDispatch.PassengerNonStationaryArrivalEvent object at 0x118032d50> Current time: 504.2135110856258,
 Post-event state below
Source destination queue matrix count
 [[0 41 35 42 24 35 35 30 30]
 [3 0 0 0 2 0 3 5 4]
 [4 4 0 2 3 1 4 0 3]
 [4 2 1 0 5 1 0 2 0]
 [3 3 3 1 0 1 1 1 6]
 [3 3 1 2 3 0 1 2 3]
 [0 3 5 3 3 2 0 2 2]
 [1 3 3 2 2 8 0 0 4]
 [1 0 3 1 3 1 4 1 0]]
Source destination matrix
 [[0 41.0 35.0 42.0 24.0 35.0 35.0 30.0 10.0]
 [3 0 0 0 2.0 0 3.0 4.0 4.0]
 [4 4 0 2.0 3.0 1.0 4.0 0 3.0]
 [4 2 1 0 5.0 1.0 0 2.0 0]
 [3 3 3 1 0 1.0 1.0 1.0 6.0]
 [3 3 1 2 3 0 1.0 2.0 3.0]
 [0 3 5 3 3 2 0 2.0 2.0]
 [1 3 3 2 2 8 0 0 4.0]
 [1 0 3 1 3 1 4 1 0]]
Car 1 - onboard:[0, 0, 0, 0, 0, 0, 0, 0, 0] floor: 2 next floor: 0 status: 1 direction: 0 final dest: 8
Car 2 - onboard:[0, 0, 0, 0, 0, 20, 0, 0, 0] floor: 0 next floor: 5 status: 1 direction: 1 final dest: 5
[array([[ 0., 20.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]]),
 array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])]
```

As shown, car 1 is at capacity for allocations with 20 in its request array (2$^{\text{nd}}$ from bottom). Car 2 is also at capacity with 20 onboard (3$^{\text{rd}}$ array from bottom).

3. Car dynamics is correct

   a. Travel time, passenger transfer time, door open/close time

4. Passenger logic is correct

   a. Non-stationary arrival rates are correct based on schedule and building population

A3. Additional Figures

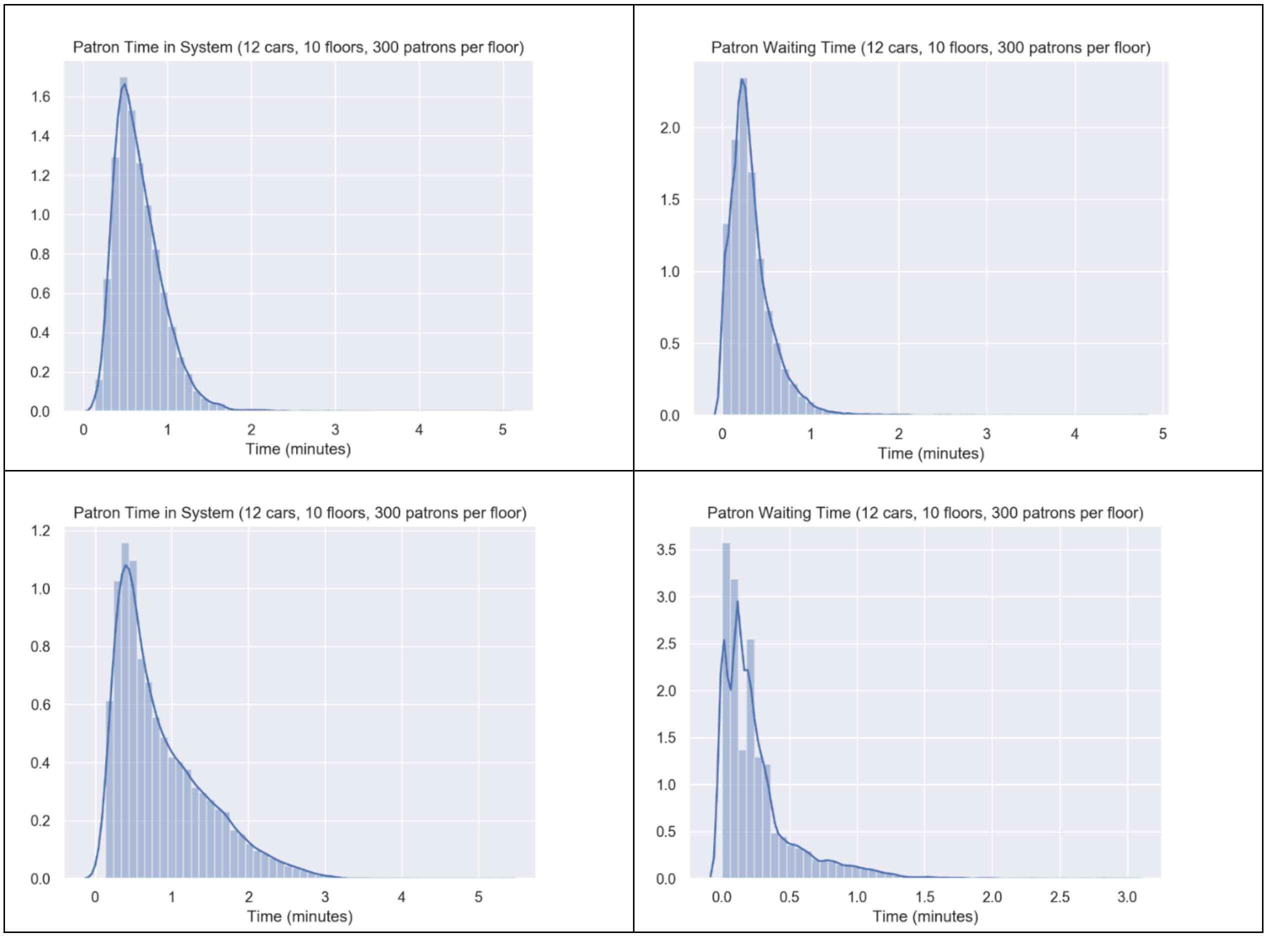

Figure. Distribution of journey time (left), and waiting time (right) for DD (top) and HUFF (bottom)

A4. Source Code

See https://github.com/do-ryan/destination-dispatch-elevator-simulation

replication.py

```
import pprint
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from custom_library import FunctionalEventNotice, DTStatPlus, FIFOQueuePlus,
CI_95
from elevator_car_traditional import ElevatorCarTraditional, Passenger
from elevator_car_dest_dispatch import ElevatorCarDestDispatch

import pythonsim.SimFunctions as SimFunctions
import pythonsim.SimRNG as SimRNG
import pythonsim.SimClasses as SimClasses

import seaborn as sns
import shutil
import os
sns.set()

pp = pprint.PrettyPrinter()

np.random.seed(1)

ZSimRNG = SimRNG.InitializeRNSeed()
s = 1  # constant s as parameter to sim


class ReplicationTraditional:
    def __init__(self,
                 run_length=120.0,
                 warm_up=0,
                 num_floors=12,
                 pop_per_floor=100,
                 num_cars=2,
                 car_capacity=20,
                 mean_passenger_interarrival=0.2,
                 write_to_csvs=False
                 ):
        """
        Models the operation of an elevator system for one replication. The
elevator assignment logic also resides here.
```

```
        Args:
            All time units are in minutes.
        """
        self.run_length = run_length
        self.warm_up = warm_up
        self.num_floors = num_floors
        self.num_cars = num_cars
        assert self.num_floors > 1

        self.mean_passenger_interarrival = mean_passenger_interarrival  # for
stationary option
        self.pop_per_floor = pop_per_floor

        # daily patterns for non-stationary. Values are % of building population
per 5 minutes, by hour starting 12AM
        self.upbound_arrival_rate = np.array(
            [0, 0, 0, 0, 0, 0, 0, 0, 8, 0.125, 0.125, 0.125, 2.75, 5.5, 0.125,
0.125, 0.125, 0.125, 0, 0, 0, 0, 0, 0])
        self.downbound_arrival_rate = np.array(
            [0, 0, 0, 0, 0, 0, 0, 0, 0.125, 0.125, 0.125, 0.125, 5.5, 2.75,
0.125, 0.125, 0.125, 8, 0, 0, 0, 0, 0, 0])
        self.crossfloor_arrival_rate = \
            np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0])

        self.write_to_csvs = write_to_csvs
        if self.write_to_csvs:
            self.figure_dir = '../figures/data'
            shutil.rmtree(self.figure_dir)
            os.mkdir(self.figure_dir)
        else:
            self.figure_dir = None

        self.floor_queues = [FIFOQueuePlus(id=i, figure_dir=self.figure_dir)
                             for i, _ in enumerate(range(0, self.num_floors))]

        self.Calendar = SimClasses.EventCalendar()
        self.WaitingTimes = DTStatPlus()
        self.TimesInSystem = DTStatPlus()
        self.TravelTimes = DTStatPlus()
        self.AllStats = [self.WaitingTimes, self.TimesInSystem, self.TravelTimes]
        self.AllArrivalTimes = []

        self.cars = [ElevatorCarTraditional(outer=self, capacity=car_capacity)
for _ in range(0, self.num_cars)]
        SimClasses.Clock = 0
        SimRNG.ZRNG = SimRNG.InitializeRNSeed()
        # don't need SimFunctionsInit .. using separate replication instances

    def __call__(self, print_trace: bool):
        return self.main(print_trace)
```

I

```
    class AssignRequestEvent(FunctionalEventNotice):
        def __init__(self, new_passenger: Passenger, *args, **kwargs):
            super().__init__(*args, **kwargs)
            self.new_passenger = new_passenger

        def event(self):
            assigned_car = None
            best_suitability = 0
            request_floor = self.new_passenger.source_floor
            # below, sometimes a car may not have a next floor but is not idle
either. This occurs when a car is idle
            # and a passenger arrives on its current floor.
            for car in self.outer.cars:
                suitability = 0
                if car.requests[request_floor, int(self.new_passenger.direction)]
== 1:
                    # if car is already allocated there
                    suitability = 5
                elif car.status == 0 and best_suitability <= 4:  # if car is idle
                    assert car.next_floor is None
                    suitability = 4
                elif (request_floor > (car.next_floor or car.floor) and
car.direction)\
                        or (request_floor < (car.next_floor or car.floor) and not
car.direction):
                    # if this is an incoming car
                    if self.new_passenger.direction == car.direction and
best_suitability < 3:
                        # if intended direction is same
                        suitability = 3
                    elif self.new_passenger.direction != car.direction and
best_suitability < 2:
                        # if intended direction is different
                        suitability = 2
                else:
                    suitability = 1
                # determine suitability of this car

                if suitability > best_suitability:
                    assigned_car = car
                    best_suitability = suitability
                # update most suitable so far

                elif suitability == best_suitability:
                    if abs(request_floor - (car.next_floor or car.floor)) \
                            < abs(request_floor - (assigned_car.next_floor or
assigned_car.floor)):
                        assigned_car = car
                    # if this car's suitability is tied with best so far, use
proximity as tie breaker
                    # if proximities tie, first index wins.
```

```
            assert assigned_car is not None
            assigned_car.requests[request_floor,
int(self.new_passenger.direction)] = 1

            # trigger an action from the idle vehicle. otherwise it will stay
idle indefinitely
            if assigned_car.status == 0:
                assigned_car.status = 3  # in decision process status. must not
keep as 0
                assigned_car.next_action()

    class PassengerStationaryArrivalEvent(FunctionalEventNotice):
        """Simple uniformly distributed source and destination floor, stationary
poisson."""

        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)

        def event(self):
            source = math.floor(SimRNG.Uniform(0, self.outer.num_floors, s))
            while True:
                destination = math.floor(SimRNG.Uniform(0, self.outer.num_floors,
s))
                if destination != source:
                    break
            new_passenger = Passenger(source_floor=source,
destination_floor=destination)
            self.outer.floor_queues[self.floor_queue_index].Add(new_passenger)
            self.outer.AllArrivalTimes.append(SimClasses.Clock)

            self.outer.Calendar.Schedule(
                self.outer.PassengerArrivalEvent(

EventTime=SimRNG.Expon(self.outer.mean_passenger_interarrival, 1),
                    outer=self.outer))

self.outer.Calendar.Schedule(self.outer.AssignRequestEvent(EventTime=0,

outer=self.outer,

new_passenger=new_passenger))

    class PassengerNonStationaryArrivalEvent(FunctionalEventNotice):
        """Non-stationary poisson arrival process based on daily patterns"""

        def __init__(self,
                     arrival_rates: np.array,
                     arrival_mode: int,
                     *args,
                     **kwargs):
            """
            Args:
```

```
            - arrival_rates - Must be a 24 length 1D array representing hourly
rate for each hour
            - arrival_mode - 0 for ground-floor up-bound, 1 for random floor
down to ground, 2 for cross-floor
            """
            super().__init__(*args, **kwargs)
            self.arrival_rates = arrival_rates
            self.arrival_mode = arrival_mode
            assert arrival_mode in [0, 1, 2]

        def nspp(self):
            max_rate = np.max(self.arrival_rates)
            possible_arrival = SimClasses.Clock + SimRNG.Expon(1 / (max_rate /
60), s)
            while SimRNG.Uniform(0, 1, 1) \
                    >= self.arrival_rates[int(possible_arrival / 60 % 24)] /
max_rate:
                possible_arrival += SimRNG.Expon(1 / (max_rate / 60), s)
            return possible_arrival - SimClasses.Clock

        def enqueue_passenger(self, passenger: Passenger):
            self.outer.floor_queues[passenger.source_floor].Add(passenger)

        def event(self):
            # set source, destination of new passenger
            if self.arrival_mode == 0:
                source = 0
                destination = math.floor(SimRNG.Uniform(1, self.outer.num_floors,
s))
            elif self.arrival_mode == 1:
                source = math.floor(SimRNG.Uniform(1, self.outer.num_floors, s))
                destination = 0
            elif self.arrival_mode == 2:
                source = math.floor(SimRNG.Uniform(1, self.outer.num_floors, s))
                while True:
                    destination = math.floor(SimRNG.Uniform(1,
self.outer.num_floors, s))
                    if destination != source or self.outer.num_floors <= 2:
                        break
            new_passenger = Passenger(source_floor=source,
destination_floor=destination)
            self.enqueue_passenger(new_passenger)
            self.outer.AllArrivalTimes.append(SimClasses.Clock)

            self.outer.Calendar.Schedule(
                self.outer.PassengerNonStationaryArrivalEvent(
                    arrival_rates=self.arrival_rates,
                    arrival_mode=self.arrival_mode,
                    EventTime=self.nspp(),
                    outer=self.outer))

self.outer.Calendar.Schedule(self.outer.AssignRequestEvent(EventTime=0,
```

```
outer=self.outer,

new_passenger=new_passenger))

    class EndSimulationEvent(FunctionalEventNotice):
        def event(self):
            pass

    def print_state(self):
        ### TRACE ######
        # pp.pprint([(e, e.EventTime, e.outer) for e in
self.Calendar.ThisCalendar])
        print(
            "Passengers waiting (source floor, destination floor)", [
                (i, p.destination_floor) for i, q in enumerate(
                    self.floor_queues) for p in q.ThisQueue])
        for i, car in enumerate(self.cars):
            print(f"Car {i+1} - onboard:{[len(floor) for floor in
car.dest_passenger_map]} floor: {car.floor} "
                  f"next floor: {car.next_floor} status: {car.status} direction:
{car.direction}")
        print([car.requests for car in self.cars])
        ################

    def callback(self):
        print(f"Mean time in system: {np.mean(self.TimesInSystem.Observations)} "
              f"Mean waiting time: {np.mean(self.WaitingTimes.Observations)} "
              f"Mean travel time: {np.mean(self.TravelTimes.Observations)}")

        arrivals_in_hours = np.array(self.AllArrivalTimes) / 60
        sns.lineplot(arrivals_in_hours, list(range(1, len(self.AllArrivalTimes) +
1)))
        # plot cumulative arrivals
        plt.xlabel("Time (24H)")
        plt.ylabel("Cumulative passenger arrivals")
        plt.xticks(range(math.floor(min(arrivals_in_hours)),
math.ceil(max(arrivals_in_hours)) + 1))
        plt.show()

        sns.distplot(self.TimesInSystem.Observations)  # plot histogram of times
in system
        plt.title(f"Patron Time in System ({self.num_cars} cars,
{self.num_floors} floors, {self.pop_per_floor} patrons per floor)")
        plt.xlabel("Time (minutes)")
        plt.show()

        sns.distplot(self.WaitingTimes.Observations)
        plt.title(f"Patron Waiting Time ({self.num_cars} cars, {self.num_floors}
floors, {self.pop_per_floor} patrons per floor)")
        plt.xlabel("Time (minutes)")
        plt.show()
```

```
        sns.distplot(self.TravelTimes.Observations)
        plt.title("Patron Travel Time")
        plt.xlabel("Time (minutes)")
        plt.show()

        if self.write_to_csvs:
            for queue in self.floor_queues:
                df =
pd.read_csv(f"{queue.figure_dir}/queue{queue.id}_lengths.csv", names=['time',
'count'])
                df.time = df.time / 60
                sns.lineplot(x='time', y='count', label=f'floor {queue.id}',
data=df)
            plt.xticks(range(math.floor(min(df.time)), math.ceil(max(df.time)) +
1))
            plt.xlabel("Time (24H)")
            plt.title('Number of Patrons Queuing for Elevators by Floor Over
Time')
            plt.legend()
            plt.show()

    def main(self, print_trace: bool):
        BuildingPopulation = self.pop_per_floor * self.num_floors
        self.Calendar.Schedule(
            self.PassengerNonStationaryArrivalEvent(
                arrival_rates=self.upbound_arrival_rate / 100 *
BuildingPopulation * 12,
                arrival_mode=0,
                EventTime=0,
                outer=self))
        self.Calendar.Schedule(
            self.PassengerNonStationaryArrivalEvent(
                arrival_rates=self.downbound_arrival_rate / 100 *
BuildingPopulation * 12,
                arrival_mode=1,
                EventTime=0,
                outer=self))
        self.Calendar.Schedule(
            self.PassengerNonStationaryArrivalEvent(
                arrival_rates=self.crossfloor_arrival_rate / 100 *
BuildingPopulation * 12,
                arrival_mode=2,
                EventTime=0,
                outer=self))
        self.Calendar.Schedule(self.EndSimulationEvent(EventTime=self.run_length,
outer=self))

        self.NextEvent = self.Calendar.Remove()

        while not isinstance(self.NextEvent, self.EndSimulationEvent):
            SimClasses.Clock = self.NextEvent.EventTime  # advance clock to start
```

```
of next event
            self.NextEvent.event()
            if print_trace:
                print(f"Executed event: {self.NextEvent} Current time:
{SimClasses.Clock}, Post-event state below")
            if print_trace:
                self.print_state()
            self.NextEvent = self.Calendar.Remove()

        for stat in self.AllStats:
            d, stat = self.apply_deletion_point(stat)
            print("Deletion index:", d, "Number of stat datapoints:",
len(stat.Observations))

        if self.write_to_csvs:
            self.callback()

        print(CI_95(self.TimesInSystem.Observations),
self.WaitingTimes.probInRangeCI95([0, 50/60]))
        return self.TimesInSystem, self.WaitingTimes, self.TravelTimes

    def apply_deletion_point(self, stat: DTStatPlus):
        """Use time in system as estimator for MSER."""
        s = 0
        q = 0
        m = len(stat.Observations)
        mser = []
        for d in range(m - 1, -1, -1):
            s += stat.Observations[d]
            q += stat.Observations[d]**2
            mser.append((q - s**2 / (m - d)) / (m - d)**2)

        mser.reverse()
        for d in range(1, len(mser)-1):
            if mser[d] <= min(mser[d - 1], mser[d + 1]):
                stat.Observations = stat.Observations[d::]
                return d, stat
        return 0, stat


class ReplicationDestDispatch(ReplicationTraditional):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.source_destination_queue_matrix = np.array([[FIFOQueuePlus(id=(i,
j), figure_dir=self.figure_dir)
                                                          for i, _ in
enumerate(range(0, self.num_floors))]
                                                         for j, _ in
enumerate(range(0, self.num_floors))])
        self.source_destination_matrix = np.frompyfunc(
            lambda x: len(
                x.ThisQueue), 1, 1)(
```

```
            self.source_destination_queue_matrix)
        self.cars = [ElevatorCarDestDispatch(outer=self) for _ in range(0,
self.num_cars)]

    class
PassengerNonStationaryArrivalEvent(ReplicationTraditional.PassengerNonStationaryA
rrivalEvent):
        def enqueue_passenger(self, passenger: Passenger):
            self.outer.source_destination_queue_matrix[passenger.source_floor,
passenger.destination_floor].Add(
                passenger)

    class AssignRequestEvent(FunctionalEventNotice):
        def __init__(self, new_passenger: Passenger, *args, **kwargs):
            super().__init__(*args, **kwargs)
            self.new_passenger = new_passenger

        def event(self):
            # trigger an action from the idle vehicle. otherwise it will stay
idle indefinitely
            car_incoming = False
            for car in self.outer.cars:
                if car.destination_dispatched ==
self.new_passenger.destination_floor \
                        and car.next_floor == self.new_passenger.source_floor and
car.status == 1:
                    car.requests[self.new_passenger.source_floor,
self.new_passenger.direction] += 1
                    car_incoming = True
                    break

            if not car_incoming:

self.outer.source_destination_matrix[self.new_passenger.source_floor,

self.new_passenger.destination_floor] += 1
                for car in self.outer.cars:
                    if car.status == 0:
                        car.status = 3
                        car.next_action()
                        break

    def print_state(self):
        ### TRACE ######
        # pp.pprint([(e, e.EventTime, e.outer) for e in
self.Calendar.ThisCalendar])
        print("Source destination queue matrix count\n",
              np.frompyfunc(lambda x: len(x.ThisQueue), 1,
1)(self.source_destination_queue_matrix))
        print("Source destination matrix\n", self.source_destination_matrix)
        for i, car in enumerate(self.cars):
            print(f"Car {i+1} - onboard:{[len(floor) for floor in
```

```
car.dest_passenger_map]} "
                  f"floor: {car.floor} next floor: {car.next_floor} status:
{car.status} "
                  f"direction: {car.direction} final dest:
{car.destination_dispatched}")
        pp.pprint([car.requests for car in self.cars])
        ###############

    def callback(self):
        print(f"Mean time in system: {np.mean(self.TimesInSystem.Observations)} "
              f"Mean waiting time: {np.mean(self.WaitingTimes.Observations)} "
              f"Mean travel time: {np.mean(self.TravelTimes.Observations)}")

        arrivals_in_hours = np.array(self.AllArrivalTimes) / 60
        sns.lineplot(arrivals_in_hours, list(range(1, len(self.AllArrivalTimes) +
1)))
        # plot cumulative arrivals
        plt.xlabel("Time (24H)")
        plt.ylabel("Cumulative passenger arrivals")
        plt.xticks(range(math.floor(min(arrivals_in_hours)),
math.ceil(max(arrivals_in_hours)) + 1))
        plt.show()

        sns.distplot(self.TimesInSystem.Observations)  # plot histogram of times
in system
        plt.title(f"Patron Time in System ({self.num_cars} cars,
{self.num_floors} floors, {self.pop_per_floor} patrons per floor)")
        plt.xlabel("Time (minutes)")
        plt.show()

        sns.distplot(self.WaitingTimes.Observations)
        plt.title(f"Patron Waiting Time ({self.num_cars} cars, {self.num_floors}
floors, {self.pop_per_floor} patrons per floor)")
        plt.xlabel("Time (minutes)")
        plt.show()

        sns.distplot(self.TravelTimes.Observations)
        plt.title("Patron Travel Time")
        plt.xlabel("Time (minutes)")
        plt.show()

        if self.write_to_csvs:
            curr_floor = 0
            df = pd.DataFrame(columns=['time', 'count'])
            for i, queue in
enumerate(np.nditer(self.source_destination_queue_matrix, flags=["refs_ok"])):
                if queue.item().id[0] != queue.item().id[1]:
                    if i // self.num_floors == curr_floor:
                        df = pd.concat([df,
pd.read_csv(f"{queue.item().figure_dir}/queue{queue.item().id}_lengths.csv",
                                                        names=['time',
'count'])])
```

```
                    else:
                        df.time = df.time / 60
                        df = df.astype({'count': 'int64'})
                        sns.lineplot(x='time', y='count', label=f'floor
{curr_floor}', data=df, legend=False)
                        curr_floor = i // self.num_floors
                        df =
pd.read_csv(f"{queue.item().figure_dir}/queue{queue.item().id}_lengths.csv",
                                        names=['time', 'count'])
            df.time = df.time / 60
            df = df.astype({'count': 'int64'})
            sns.lineplot(x='time', y='count', label=f'floor {curr_floor}',
data=df, legend=False)
            # graph queue sizes over time by floor
            plt.xticks(range(math.floor(min(df.time)), math.ceil(max(df.time)) +
1))
            plt.xlabel("Time (24H)")
            plt.title('Number of Patrons Queuing for Elevators by Floor Over Time
(Destination Dispatch)')
            plt.legend(loc='upper left')
            plt.show()

print(ReplicationTraditional(run_length= 60*24,num_floors=5, pop_per_floor=100,
num_cars=2, write_to_csvs=False)(print_trace=True)[1].probInRangeCI95([0,
50/60]))
```

elevator_car_traditional.py

```
from custom_library import FunctionalEventNotice

import pythonsim.SimClasses as SimClasses
import numpy as np
import itertools


class ElevatorCarTraditional(SimClasses.Resource):
    def __init__(self,
                 outer, # must be ReplicationTraditional class
                 initial_floor=0,
                 capacity=20,
                 door_move_time=0.0417,  # 2.5 seconds
                 passenger_move_time=0.0167,  # 1 second
                 acceleration=1.0,  # m/s^2
                 top_speed=3.0,  # m/s
                 floor_distance=4.5  # metres
                 ):
        self.outer = outer

        # resource states
        self.status = 0  # 0 for idle car, 1 for moving, 2 for transferring, 3
for in process of choosing next task
        # only changes at the start of some action
        self.Busy = 0
        self.NumberOfUnits = capacity
        self.NumBusy = SimClasses.CTStat()
        self.Calendar = self.outer.Calendar

        # elevator system states
        self.floor_queues = self.outer.floor_queues
        self.floor = initial_floor
        self.next_floor = None
        self.dest_passenger_map = [[] for _ in range(len(self.floor_queues))]  #
[destination floor][list_of_passengers]
        self.requests = np.zeros(shape=(len(self.floor_queues), 2))
        # for each floor: index 0, 1 is down, up request respectively. value of 0
indicates no request.
        self.direction = None
        # direction is only updated when car goes idle, or when car starts moving
from idle.

        # kinematic properties
        self.door_move_time = door_move_time
        self.passenger_move_time = passenger_move_time
        self.acceleration = acceleration
        self.top_speed = top_speed
        self.floor_distance = floor_distance
```

```
        # Note: must change status as soon as start action event is scheduled.
        # Otherwise it allows task allocated to trigger another action event if
car is idle

    def floor_dwell(self, num_passengers: int):
        """Return amount of time to spend transferring passengers. Door open/
close is lumped into move event."""
        return self.passenger_move_time * num_passengers

    class PickupEvent(FunctionalEventNotice):
        def event(self):
            """Pick-up as many passengers as possible from current floor, update
self resource, add waiting time data."""
            self.outer.status = 2
            num_passengers = 0
            directions_requested =
set(np.nonzero(self.outer.requests[self.outer.floor])[0].tolist())
            num_to_pick_up = len(
                [p for p in self.outer.floor_queues[self.outer.floor].ThisQueue
if p.direction in directions_requested])
            while num_passengers < num_to_pick_up and self.outer.Busy <
self.outer.NumberOfUnits:
                next_passenger =
self.outer.floor_queues[self.outer.floor].Remove()
                if self.outer.requests[self.outer.floor,
next_passenger.direction]:
                    # if this car has a request equal to the first person in
queue's intended direction
                    assert isinstance(next_passenger, Passenger)
                    self.outer.Seize(1)

self.outer.dest_passenger_map[next_passenger.destination_floor].append(next_passe
nger)
                    self.outer.outer.WaitingTimes.Record(SimClasses.Clock -
next_passenger.CreateTime)
                    num_passengers += 1
                else:
                    # if the car does not have a request equal to first person's
intended
                    # direction, put him to back of line

self.outer.floor_queues[self.outer.floor].ThisQueue.append(next_passenger)

            self.outer.requests[self.outer.floor, 0] = 0
            self.outer.requests[self.outer.floor, 1] = 0
            # ensure that there are no more requests on this floor for this car
otherwise it will infinitely
            # try to pick them up if the car is full.


self.outer.Calendar.Schedule(self.outer.PickupEndEvent(EventTime=self.outer.floor
_dwell(num_passengers),
```

```
outer=self.outer))

            for passenger in self.outer.floor_queues[self.outer.floor].ThisQueue:
                # Guaranteed O(2) per pickup.
                if passenger.direction in directions_requested:
                    directions_requested -= {passenger.direction}
                    # for all passengers requests that were allocated to this car
that couldn't get on,
                    # re-allocate car to those requests

self.outer.Calendar.Schedule(self.outer.outer.AssignRequestEvent(new_passenger=pa
ssenger,

EventTime=0,

outer=self.outer.outer))
                if len(directions_requested) == 0:
                    break

    class PickupEndEvent(FunctionalEventNotice):
        def event(self):
            for dest in self.outer.dest_passenger_map:
                for passenger in dest:
                    passenger.entry_time = SimClasses.Clock
            self.outer.next_action()

    class DropoffEvent(FunctionalEventNotice):
        def event(self):
            """Drop-off all passengers on self with destination as current floor.
Add time"""
            self.outer.status = 2
            num_passengers = 0
            while len(self.outer.dest_passenger_map[self.outer.floor]) > 0:
                self.outer.Free(1)
                next_passenger =
self.outer.dest_passenger_map[self.outer.floor].pop(0)
                self.outer.outer.TimesInSystem.Record(SimClasses.Clock -
next_passenger.CreateTime)
                self.outer.outer.TravelTimes.Record(SimClasses.Clock -
next_passenger.entry_time)
                num_passengers += 1
            self.outer.Calendar.Schedule(
                self.outer.DropoffEndEvent(
                    EventTime=self.outer.floor_dwell(num_passengers),
                    outer=self.outer))

    class DropoffEndEvent(FunctionalEventNotice):
        def event(self):
            self.outer.next_action()

    class MoveEvent(FunctionalEventNotice):
```

```
        def __init__(self, destination_floor: int, *args, **kwargs):
            super().__init__(*args, **kwargs)
            self.destination_floor = destination_floor

        def event(self):
            """Return amount of time required to move from current floor to
destination floor after doors close."""
            time_to_top_speed = self.outer.top_speed / self.outer.acceleration
            distance_to_top_speed = self.outer.acceleration *
time_to_top_speed**2 / 2
            total_distance = (abs(self.destination_floor - self.outer.floor) *
self.outer.floor_distance)
            self.outer.next_floor = self.destination_floor
            if distance_to_top_speed > total_distance / 2:
                travel_time = (total_distance / self.outer.acceleration) ** (1 /
2)
            else:
                travel_time = time_to_top_speed \
                    + (total_distance - (2 * distance_to_top_speed)) /
self.outer.top_speed \
                    + time_to_top_speed
            self.outer.Calendar.Schedule(
                self.outer.MoveEndEvent(
                    EventTime=self.outer.door_move_time +
                    travel_time /
                    60 +
                    self.outer.door_move_time,
                    outer=self.outer))

            self.outer.direction = int(self.outer.next_floor > self.outer.floor)
            self.outer.status = 1

    class MoveEndEvent(FunctionalEventNotice):
        def event(self):
            self.outer.floor = self.outer.next_floor
            # on MoveEnd, car.next_floor = car.floor
            self.outer.next_action()

    def next_action(self):
        """Triggered after moving, or done a transfer. Schedules event
representing next action."""
        if np.sum(self.requests) == 0 and len(
                [passenger for passenger in
itertools.chain.from_iterable(self.dest_passenger_map)]) == 0:
            # if no requests nor drop-offs, go idle. this is the only case where
the car should go idle.
            # if the car goes idle while it has a potential task, a new task may
trigger a state of multiple actions,
            # because new passengers look for idle cars.
            self.status = 0
            self.next_floor = None
            self.direction = None
```

```
        else:
            if len(self.dest_passenger_map[self.floor]) > 0:
                self.Calendar.Schedule(self.DropoffEvent(EventTime=0,
outer=self))
                self.outer.status = 2
            elif np.sum(self.requests[self.floor, :]) > 0:
                self.Calendar.Schedule(self.PickupEvent(EventTime=0, outer=self))
                self.outer.status = 2
            else:  # if no more pickups nor drop-offs on current floor, take on
the next request
                scan_start = self.floor  # scan start is the floor that "up-
bound" and "down-bound" are relative to
                if self.status == 3:
                    #  if in process, take the next request if there is one-
start closer to terminal ends.
                    # currently there is no direction so we need to decide which
direction this car will proceed in and
                    # which request to start on.
                    if np.sum(self.requests[:, 1]) \
                            + len([queue for queue in
itertools.chain.from_iterable(self.dest_passenger_map)])\
                            > np.sum(self.requests[:, 0]) \
                            + len([queue for queue in
itertools.chain.from_iterable(self.dest_passenger_map)]):
                        #  if there are more up-bound requests than down-bound,
go up-bound.
                        self.direction = 1
                        scan_start = 0  # go to globally lowest up-bound request
                    else:
                        self.direction = 0
                        scan_start = self.requests.shape[0] - 1
                        # go to globally highest down-bound request

                # from here downwards, if car is currently idle, then it will
have chosen a direction.
                # if idle,
                # it will start at the globally lowest upbound task, or highest
downbound task, dropping off passengers
                # on the car along the way.
                # if the car already has a direction, it will sequentially take
tasks in that direction until all
                # tasks are finished in that direction.
                if self.direction == 1:
                    # if going upwards
                    if np.max(self.requests[:, 1][scan_start:]) > 0:
                        # if there are up-bound requests above
                        next_pickup = np.argmax(self.requests[:, 1][scan_start:])
+ scan_start
                    else:
                        next_pickup = None
                    if len([queue for queue in
itertools.chain.from_iterable(self.dest_passenger_map[scan_start:])]):
```

```
                        next_dropoff = [queue for queue in
itertools.chain.from_iterable(

self.dest_passenger_map[scan_start:])][0].destination_floor
                        # if there are drop-offs above
                    else:
                        next_dropoff = None

                    if next_pickup is not None and next_dropoff is not None:
                        next_destination = min(next_pickup, next_dropoff)
                    elif next_pickup is not None or next_dropoff is not None:
                        if next_pickup is None:
                            next_destination = next_dropoff
                        else:
                            next_destination = next_pickup
                    # from pool of allocated up-bound requests or drop-offs,
                    # pick the lowest floor that is higher than current floor
                    if next_pickup is None and next_dropoff is None:
                        # if there are no up-bound requests of drop-offs above,
take next request
                        self.status = 3
                        self.next_floor = None
                        self.direction = None
                        self.next_action()
                    else:

self.Calendar.Schedule(self.MoveEvent(destination_floor=int(next_destination),
                                                              EventTime=0,
                                                              outer=self))
                else:  # if going downwards or currently directionless
(previously idle, and just picked-up)
                    if np.max(self.requests[:, 0][0:scan_start + 1]) > 0:
                        # if there are down-bound requests below
                        next_pickup = scan_start -
np.argmax(np.flip(self.requests[0:scan_start + 1, 0], axis=0))
                        # take the highest down-bound request below
                    else:
                        next_pickup = None
                    if len([queue for queue in
itertools.chain.from_iterable(self.dest_passenger_map[0:scan_start + 1])]):
                        next_dropoff = [queue for queue in
itertools.chain.from_iterable(
                            self.dest_passenger_map[0:scan_start + 1])][-
1].destination_floor
                        # if there are drop-offs below, save the highest floor
drop-off
                    else:
                        next_dropoff = None

                    if next_pickup is not None and next_dropoff is not None:
                        next_destination = max(next_pickup, next_dropoff)
                    elif next_pickup is not None or next_dropoff is not None:
```

```
                        next_destination = next_pickup or next_dropoff
                    # from pool of allocated down-bound requests or drop-offs,
                    # pick the lowest floor that is higher than current floor

                    if next_pickup is None and next_dropoff is None:
                        # if there are no down-bound requests or drop-offs below,
take next request
                        self.status = 3
                        self.next_floor = None
                        self.direction = None
                        self.next_action()
                    else:

self.Calendar.Schedule(self.MoveEvent(destination_floor=int(next_destination),
                                                              EventTime=0,
                                                              outer=self))


class Passenger(SimClasses.Entity):
    def __init__(self,
                 source_floor: int,
                 destination_floor: int,
                 ):
        super().__init__()
        self.source_floor = source_floor
        self.destination_floor = destination_floor
        self.direction = int(self.destination_floor > self.source_floor)
        self.entry_time = None
```

elevator_car_dest_dispatch.py

```
from custom_library import FunctionalEventNotice
from elevator_car_traditional import ElevatorCarTraditional

import pythonsim.SimClasses as SimClasses
import numpy as np
import itertools
import math


class ElevatorCarDestDispatch(ElevatorCarTraditional):
    def __init__(self, outer, max_wait_threshold=3, *args, **kwargs):
        """
        Args:
            - outer: ReplicationDestDispatch
            - max_wait_threshold: if anyone waits past this threshold, they get
priority.
        """
        super().__init__(outer, *args, **kwargs)
        self.source_destination_matrix = self.outer.source_destination_matrix
        self.source_destination_queue_matrix =
self.outer.source_destination_queue_matrix
        self.destination_dispatched = None  # the destination floor this car is
dispatched to
        self.max_wait_threshold = max_wait_threshold

    class PickupEvent(FunctionalEventNotice):
        def event(self):
            """Pick-up as many passengers going to current destination as
possible from current floor,
            update self resource, add waiting time data."""
            self.outer.status = 2
            num_passengers = 0
            directions_requested =
np.nonzero(self.outer.requests[self.outer.floor])[0].tolist()
            assert len(directions_requested) == 1
            direction_requested = directions_requested[0]
            leftovers = []
            while (self.outer.requests[self.outer.floor, direction_requested] > 0
                    or self.outer.source_destination_matrix[self.outer.floor,
self.outer.destination_dispatched] > 0):
                # while this car still need to pick up allocated tasks or if
there are new ones
                next_passenger =
self.outer.source_destination_queue_matrix[self.outer.floor,

self.outer.destination_dispatched].Remove()
                # O(number of queued passengers/ num_floors)

                if self.outer.requests[self.outer.floor, direction_requested] <=
```

```
0:
                    self.outer.source_destination_matrix[self.outer.floor,
self.outer.destination_dispatched] -= 1
                    # if the car is now picking up unallocated requests, deplete
from the matrix.
                if self.outer.Busy < self.outer.NumberOfUnits:
                    self.outer.Seize(1)

self.outer.dest_passenger_map[next_passenger.destination_floor].append(next_passe
nger)
                    self.outer.outer.WaitingTimes.Record(SimClasses.Clock -
next_passenger.CreateTime)
                    num_passengers += 1
                else:

self.outer.Calendar.Schedule(self.outer.outer.AssignRequestEvent(new_passenger=ne
xt_passenger,

EventTime=0,

outer=self.outer.outer))
                    leftovers.append(next_passenger)
                    # if there are more suitable passengers than space available.
usually if
                    # too many are allocated in AssignRequestEvent
                self.outer.requests[self.outer.floor, direction_requested] -= 1
                # subtract from requests array whether or not the passsenger is
allocated or non-allocated.

            self.outer.requests[self.outer.floor, direction_requested] = 0
            # must be 0 here. if there were leftover requests, there are now in
leftovers. might have been negative
            # if there were rejects (more waiting passengers than expected).

            self.outer.source_destination_queue_matrix[self.outer.floor,
self.outer.destination_dispatched].ThisQueue \
                = leftovers +
self.outer.source_destination_queue_matrix[self.outer.floor,

self.outer.destination_dispatched].ThisQueue
                # make sure people added back to the queue are put in the front

self.outer.Calendar.Schedule(self.outer.PickupEndEvent(EventTime=self.outer.floor
_dwell(num_passengers),

outer=self.outer))

    def next_action(self):
        """Triggered after moving, or done a transfer. Schedules event
representing next action."""
        if np.sum(self.requests) == 0 \
                and len([passenger for passenger in
```

```
itertools.chain.from_iterable(self.dest_passenger_map)]) == 0:
            # If no requests nor dropoffs, check central pool for waiting
requests.
            # Pick destination-direction combination with highest count of
passengers.
            largest_count = 0
            exceed_max_wait_threshold = False
            chosen_destination_direction = ()
            chosen_destination_longest_wait = 0
            for i in range(2 * self.source_destination_matrix.shape[1]):
                destination_direction = (i // 2, i % 2)
                if destination_direction[1] == 1:
                    sum = np.sum(self.source_destination_matrix[
                                 0:destination_direction[0] + 1,
                                 destination_direction[0]])
                else:
                    sum = np.sum(self.source_destination_matrix[
                                 destination_direction[0]::,
                                 destination_direction[0]
                                 ])
                # Find count of passengers with this destination and direction

                destination_longest_wait = 0
                if destination_direction[1] == 0:
                    sweep =
self.source_destination_queue_matrix[destination_direction[0]+1::,

destination_direction[0]]
                else:
                    sweep =
self.source_destination_queue_matrix[0:destination_direction[0],

destination_direction[0]]

                for target_destination_queue in sweep:
                    # O(num floors)
                    if target_destination_queue.ThisQueue:
                        if sum >= largest_count:
                            destination_longest_wait = max(
                                destination_longest_wait,
                                SimClasses.Clock -
target_destination_queue.ThisQueue[0].CreateTime)
                        # Find earliest arrival time out of all waiting to go to
this destination in this direction
                if destination_longest_wait >= self.max_wait_threshold:
                        exceed_max_wait_threshold = True
                            # But if any queue has a person waiting past the
allowed threshold,
                        # here on out prioritize only floors with people waiting
past that threshold.

                if destination_longest_wait >= self.max_wait_threshold or not
```

```
exceed_max_wait_threshold:
                    # if max wait threshold has not yet been exceeded, consider
all. otherwise only consider those past threshold.
                    if sum > largest_count:
                        largest_count = sum
                        chosen_destination_direction = destination_direction
                        chosen_destination_longest_wait =
destination_longest_wait
                    elif sum == largest_count:
                        if destination_longest_wait >
chosen_destination_longest_wait:
                            chosen_destination_longest_wait =
destination_longest_wait
                            chosen_destination_direction = destination_direction
                    # Choose the destination/direction with the largest count.
                    # In the case of a tie, choose the one with the longer wait

            if largest_count > 0:
                # in the state where all combinations are WIP (all negative
count) and this car has no tasks,
                # this condition prevents this car from taking other cars' tasks.
                self.destination_dispatched = chosen_destination_direction[0]
                if chosen_destination_direction[1] == 1:
                    source_floors = np.nonzero(self.source_destination_matrix[
                        0:chosen_destination_direction[0] + 1,
                        chosen_destination_direction[0]])[0]
                else:
                    source_floors = np.nonzero(self.source_destination_matrix[
                                              chosen_destination_direction[0]::,

chosen_destination_direction[0]])[0] + chosen_destination_direction[0]
                for floor in source_floors:
                    num_more_to_pickup = min((self.NumberOfUnits - self.Busy) -
np.sum(self.requests),

self.source_destination_matrix[floor, chosen_destination_direction[0]])
                    self.requests[floor, chosen_destination_direction[1]] +=
num_more_to_pickup
                    self.source_destination_matrix[floor,
chosen_destination_direction[0]] -= num_more_to_pickup
                    # The source-destination matrix acts as a global tracker of
number of waiting passengers
                    # without allocated car. This updates the matrix when car
allocates itself.
                    # This also makes sure the car is not allocating itself to
more than it can carry.

        super().next_action()
```

experiment.py

```
from replication import ReplicationDestDispatch, ReplicationTraditional

import numpy as np
import math
import sys
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()


class Experiment:
    def __init__(self):
        self.floor_counts = [3, 5, 10, 15, 20, 25, 30]
        self.floor_pops = [25, 50, 100, 200, 300]
        self.fleet_sizes = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 18,
20, 25, 30, 35, 40]
        self.base_path = '../experiments'

    def __call__(self):
        return self.main()

    def fleet_sizing_by_prob_threshold(self,
                                       replication_class,
                                       num_floors,
                                       pop_per_floor,
                                       target=50 / 60,
                                       threshold=0.95,
                                       min_fleet_size=0,
                                       max_fleet_size=100,
                                       output_index=1):
        if min_fleet_size == max_fleet_size:
            return max_fleet_size
        mid = (max_fleet_size + min_fleet_size) // 2
        replication = replication_class(run_length=60 * 24 * 2,
                                        num_floors=num_floors,
                                        pop_per_floor=pop_per_floor,
                                        num_cars=mid)
        output_stats = replication.main(print_trace=False)  # 2 is waiting time
        output = output_stats[output_index]
        print(f"fleet size: {mid}, prob: {output.probInRangeCI95([0, target])}")
        if output.probInRangeCI95([0, target])[1][0] >= threshold:
            return
self.fleet_sizing_by_prob_threshold(min_fleet_size=min_fleet_size,
                                                       max_fleet_size=mid,

replication_class=replication_class,
                                                       num_floors=num_floors,

pop_per_floor=pop_per_floor)
```

```
        else:
            return self.fleet_sizing_by_prob_threshold(min_fleet_size=mid + 1,

max_fleet_size=max_fleet_size,

replication_class=replication_class,
                                                       num_floors=num_floors,

pop_per_floor=pop_per_floor)

    # def fleet_sizing_by_prob_threshold(self,
    #                                    replication_class,
    #                                    num_floors,
    #                                    pop_per_floor,
    #                                    target=50 / 60,
    #                                    threshold=0.95,
    #                                    output_index=1):
    #     """Iterative algorithm O(n)"""
    #     fleet_sizes = self.fleet_sizes
    #     for fleet_size in fleet_sizes:
    #         replication = replication_class(run_length=60 * 24 * 2,
    #                                         num_floors=num_floors,
    #                                         pop_per_floor=pop_per_floor,
    #                                         num_cars=fleet_size)
    #         output_stats = replication.main(print_trace=False)  # 2 is waiting
time
    #         output = output_stats[output_index]
    #         print(f"fleet size: {fleet_size}, prob: {output.probInRangeCI95([0,
target])}")
    #         if output.probInRangeCI95([0, target])[1][0] >= threshold:
    #             return fleet_size

    @staticmethod
    def fleet_sizing_by_best_selection(replication_class,
                                       num_floors,
                                       pop_per_floor,
                                       fleet_sizes=[2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 15, 20],
                                       alpha=0.05,
                                       delta=0.25):  # delta of 0.0005 seems to
give me an indifference zone of 5s...
        """Unused"""
        n_0 = math.inf
        Y = []
        I = fleet_sizes
        for fleet_size in I:
            replication = replication_class(run_length=60 * 24 * 2,
                                            num_floors=num_floors,
                                            pop_per_floor=pop_per_floor,
                                            num_cars=fleet_size)
            TimesInSystem, _, _ = replication.main(print_trace=False)
            n_0 = min(len(TimesInSystem.Observations), n_0)
```

```
            Y.append(TimesInSystem)
        n_0 = 20  # allow room for iteration when r gets incremented
        eta = 0.5 * ((2 * alpha / (len(I) - 1))**(-2 / (n_0 - 1)) - 1)
        t2 = 2 * eta * (n_0 - 1)
        s2 = []
        for i in range(len(I)):
            print("step 2 i:", i)
            s2.append([])
            for h in range(len(I)):
                if h <= i:
                    s2[-1].append(None)
                else:
                    sum = np.sum(np.square(np.array(Y[i].Observations[0:n_0]) -
np.array(Y[h].Observations[0:n_0])
                                               - (np.mean(Y[i].Observations[0:n_0]) -
np.mean(Y[h].Observations[0:n_0]))))
                    s2[-1].append(sum / (n_0 - 1))
        r = n_0
        while len(I) > 3:
            print("I:", I)
            I_old = I
            I = []
            for i in range(len(I_old)):
                print("step 3 i:", i)
                include = True
                for h in range(len(I_old)):
                    if h <= i:
                        pass
                    else:
                        W_ih = max(0, (delta / (2 * r)) * (t2 * s2[min(i,
h)][max(i, h)] / delta**2 - r))
                        if np.mean(Y[i].Observations[0: r]) >
(np.mean(Y[h].Observations[0:r]) + W_ih):
                            include = False
                if include:
                    I.append(I_old[i])
            r += 1

        return I

    def figures(self):
        tis_trad = np.empty(shape=(len(self.floor_counts), len(self.floor_pops)))
        tis_dd = np.empty(shape=(len(self.floor_counts), len(self.floor_pops)))
        wait_trad = np.empty(shape=(len(self.floor_counts),
len(self.floor_pops)))
        wait_dd = np.empty(shape=(len(self.floor_counts), len(self.floor_pops)))
        for i, floor_count in enumerate(self.floor_counts):
            for j, floor_pop in enumerate(self.floor_pops):
                f =
open(f"{self.base_path}/{floor_count}floors_{floor_pop}pflr.txt", "r")
                contents = f.read()
                tis, wait = tuple([(int(measure_fleetsizes.split(' ')[0]),
```

```
int(measure_fleetsizes.split(' ')[-1]))
                                   for measure_fleetsizes in
contents.split('\n')[-2:-4:-1][::-1]])
                tis_trad[i][j] = tis[0]
                tis_dd[i][j] = tis[1]
                wait_trad[i][j] = wait[0]
                wait_dd[i][j] = wait[1]

        sns.heatmap(tis_trad, xticklabels=self.floor_pops,
yticklabels=self.floor_counts, linewidths=0.25, annot=True)
        plt.title("Required number of elevator cars to achieve (3*(building
height/ top speed)+50)s\n journey time with 95% probability at 95% confidence
(Traditional Algorithm)")
        plt.xlabel("Number of patrons per floor")
        plt.ylabel("Number of floors")
        plt.show()
        sns.heatmap(tis_dd, xticklabels=self.floor_pops,
yticklabels=self.floor_counts, linewidths=0.25, annot=True)
        plt.title("Required number of elevator cars to achieve (3*(building
height/ top speed)+50)s\n journey time with 95% probability at 95% confidence
(Destination Dispatch Algorithm)")
        plt.xlabel("Number of patrons per floor")
        plt.ylabel("Number of floors")
        plt.show()
        sns.heatmap(wait_trad, xticklabels=self.floor_pops,
yticklabels=self.floor_counts, linewidths=0.25, annot=True)
        plt.title("Required number of elevator cars to achieve 50s wait time\n
with 95% probability at 95% confidence (Traditional Algorithm)")
        plt.xlabel("Number of patrons per floor")
        plt.ylabel("Number of floors")
        plt.show()
        sns.heatmap(wait_dd, xticklabels=self.floor_pops,
yticklabels=self.floor_counts, linewidths=0.25, annot=True)
        plt.title("Required number of elevator cars to achieve 50s wait time\n
with 95% probability at 95% confidence (Destination Dispatch Algorithm)")
        plt.xlabel("Number of patrons per floor")
        plt.ylabel("Number of floors")
        plt.show()
        sns.heatmap(tis_dd - tis_trad, xticklabels=self.floor_pops,
yticklabels=self.floor_counts, linewidths=0.25, annot=True)
        plt.title("Net improvement of Destination Dispatch\n for elevator car
fleet sizing measured by journey time")
        plt.xlabel("Number of patrons per floor")
        plt.ylabel("Number of floors")
        plt.show()
        sns.heatmap(wait_dd - wait_trad, xticklabels=self.floor_pops,
yticklabels=self.floor_counts, linewidths=0.25, annot=True)
        plt.title("Net improvement of Destination Dispatch\n for elevator car
fleet sizing measured by wait time")
        plt.xlabel("Number of patrons per floor")
        plt.ylabel("Number of floors")
        plt.show()
```

```
    def main(self):
        waiting_time_threshold = 50 / 60
        floor_distance = 4.5
        top_speed = 3.0

        for floor_count in self.floor_counts:
            tis_threshold = (floor_distance * floor_count / top_speed * 3) / 60 +
waiting_time_threshold
            for floor_pop in self.floor_pops:
                sys.stdout =
open(f'{self.base_path}/{floor_count}floors_{floor_pop}pflr.txt', 'w+')
                fleet_size_trad_tis = self.fleet_sizing_by_prob_threshold(
                    replication_class=ReplicationTraditional,
                    num_floors=floor_count,
                    pop_per_floor=floor_pop,
                    target=tis_threshold,
                    output_index=0)
                fleet_size_dd_tis = self.fleet_sizing_by_prob_threshold(
                    replication_class=ReplicationDestDispatch,
                    num_floors=floor_count,
                    pop_per_floor=floor_pop,
                    target=tis_threshold,
                    output_index=0)
                fleet_size_trad_wait = self.fleet_sizing_by_prob_threshold(
                    replication_class=ReplicationTraditional,
                    num_floors=floor_count,
                    pop_per_floor=floor_pop,
                    target=waiting_time_threshold,
                    output_index=1)
                fleet_size_dd_wait = self.fleet_sizing_by_prob_threshold(
                    replication_class=ReplicationDestDispatch,
                    num_floors=floor_count,
                    pop_per_floor=floor_pop,
                    target=waiting_time_threshold,
                    output_index=1)
                print(fleet_size_trad_tis, fleet_size_dd_tis)
                print(fleet_size_trad_wait, fleet_size_dd_wait)


exp = Experiment()
# exp.main()
exp.figures()
```

custom_library.py

```
import pythonsim.SimClasses as SimClasses

from abc import abstractmethod
from pathlib import Path
import csv
import numpy as np


def CI_95(data):
    a = np.array(data)
    n = len(a)
    m = np.mean(a)
    sd = np.std(a, ddof=1)
    hw = 1.96 * sd / np.sqrt(n)
    return m, [m - hw, m + hw]


class DTStatPlus:
    def __init__(self):
        super().__init__()
        self.Observations = []

    def Record(self, X):
        self.Observations.append(X)

    def BatchedQuantile(self, b: int, q: float):
        """
        Args:
            - b: batch size
            - q: quantile
        """

        batched_observations =
np.array_split(np.random.permutation(self.Observations),
                                             indices_or_sections=b,
                                             axis=0)
        return np.array([np.quantile(batch, q) for batch in
batched_observations])

    def probInRangeCI95(self, range: list):
        sample_prob = np.mean(
            np.logical_and(
                np.array(self.Observations) >= range[0],
np.array(self.Observations) <= range[1]).astype(int))
        se = np.sqrt(sample_prob * (1 - sample_prob) / len(self.Observations))
        return sample_prob, [sample_prob - 1.96 * se, sample_prob + 1.96 * se]


class FIFOQueuePlus(SimClasses.FIFOQueue):
```

```
    def __init__(self, id, figure_dir=None):
        super().__init__()
        self.id = id
        self.figure_dir = figure_dir

    def output_size(self):
        if self.figure_dir is not None:
            with open(Path(self.figure_dir) / f'queue{self.id}_lengths.csv',
'a+', newline='') as f:
                writer = csv.writer(f)
                writer.writerow([SimClasses.Clock, self.NumQueue()])

    def Add(self,X):
        # Add an entity to the end of the queue
        self.ThisQueue.append(X)
        numqueue = self.NumQueue()
        self.WIP.Record(float(numqueue))
        self.output_size()

    def Remove(self):
        # Remove the first entity from the queue and return the object
        # after updating the queue statistics
        if len(self.ThisQueue) > 0:
            remove = self.ThisQueue.pop(0)
            self.WIP.Record(float(self.NumQueue()))
            self.output_size()
            return remove


class FunctionalEventNotice(SimClasses.EventNotice):

    def __init__(self,
                 EventTime: float,
                 outer):
        """outer refers to an outer scope class. This class is designed to be
nested in another class"""
        super().__init__()
        self.EventTime = SimClasses.Clock + EventTime
        self.outer = outer

    @abstractmethod
    def event(self):
        pass
```