Elevator Scheduling in a Multi-Story Building

James Magraken

CSC446

1 Problem Description

The objective of this project is to compare the performance of two elevator scheduling algorithms in a multi-story office building with respect to passenger wait time. The passenger wait time for a single elevator journey is defined as the sum of the time spent waiting for the elevator to arrive and the time spent onboard the elevator. My simulation considers the period from when the first passenger enters the building in the morning to when the last passenger departs in the evening. Assuming a fixed passenger input model, I investigate the mean and maximum passenger wait time over the course of a day to determine the circumstances in which each algorithm performs better.

2 Simulation Model and Parameters

2.1 Introduction to Elevator Scheduling

In a multi-story high-traffic building such as a hotel, apartment complex, or office building, efficient vertical transportation is indispensable to ensure customer satisfaction and increase worker productivity. For rapid, accessible transportation across multiple stories, elevators are often the only option. As most people can attest to, however, a poorly-designed elevator system can yield wait times so prolonged that it is quicker to travel by stairs. In some cases, insufficiently many elevators or elevators which travel too slowly are attributable to long wait times. These issues have obvious, albeit expensive, solutions. It is often impractical or even impossible to retrofit a building with faster or more elevators. The focus of this project is on another factor of elevator wait times, which are substantially cheaper and easier to modify: elevator scheduling algorithms.

Elevator scheduling refers to the order in which an elevator services requests, and in a multi-elevator system, the elevator to which a request is assigned. Unfortunately, elevator scheduling algorithms are usually proprietary and not publicly available. Moreover, many modern approaches to elevator scheduling use heuristic algorithms and machine learning to minimize wait times, which are beyond the scope of this project. However, the *elevator algorithm*, whose use

persists in many buildings today, is well-documented. Given a list of requests assigned to an individual elevator, this algorithm determines the order in which they are serviced. It is summarized as follows:

- Beginning with an idle elevator with no assigned requests, suppose the elevator receives an upwards-bound request (the case of a downwards-bound request is similar).
- Travel to the floor on which the passenger is waiting and pick them up.
- Pick up any other upwards-bound passengers en route to the original passenger's destination floor.
- Continue traveling upwards while picking up and dropping off upwards-bound passengers until the car is empty and there are no upwards-bound requests originating at or above the elevator's position.
- Travel to the origin floor of the highest downwards-bound request and repeat the previous two steps, replacing "upwards" with "downwards" and "above" with "below".
- Continue servicing requests in one direction and switching directions until all assigned requests have been fulfilled. At which point, become idle and repeat from the first step.

In this project, I consider a multi-elevator system in which each individual elevator uses the elevator algorithm. I compare two schemes for assigning an incoming request to an elevator in the system. Both schemes are examples of greedy algorithms, as they assign a request as soon as it is received based on the current state of the system, and do not modify the assignment once it is made.

The first scheme, which I refer to as the *conventional algorithm*, has many variants whose use persists to this day. The exact variant which I consider in this project is described as follows:

- When a passenger requests service, they indicate whether they wish to travel up or down by pressing one of two buttons.
- Determine the *travel distance* for each elevator to service the incoming request, and assign the request to the elevator with least travel distance.

To define the travel distance, first let e be the position of an elevator in an N-story building at the time a request is received (e may not be integral if the car is between floors), and r be the floor on which the request originates, where floors are zero-indexed (i.e., the lobby is 0). Let $d_e, d_r \in \{-1, 0, 1\}$ be the direction of the elevator and the request, respectively, where -1 means down, 0

means idle, and 1 means up. Then

$$\text{Travel distance} = \begin{cases} |r-e|, & d_e = 0 \\ |r-e|, & d_e = d_r \text{ and } d_e e \leq d_r r \\ 2N-2-|r-e|, & d_e = d_r \text{ and } d_e e > d_r r \\ 2N-2-e-r, & d_e \neq d_r \text{ and } d_e = 1 \\ e+r, & \text{otherwise} \end{cases} \tag{1}$$

In plain English, if an elevator is idle, then it can go to a new request right away, so its travel distance is just the number of floors separating it and the request. The distance is calculated similarly if an elevator is already traveling and it receives a request in the same direction en route to its destination. The first and second cases in (1) represent this scenario. If the request is in the same direction, but the elevator has already passed the origin floor of the request, then the elevator must, in the worst-case scenario, travel to the end of the building, all the way across the height of the building, then finally to the origin floor. The third case in (1) represents this situation. Finally, if the elevator's direction and the request's direction are different, then the elevator must, in the worst-case scenario, travel to the end of the building, reverse directions, then travel to the origin floor. The last two cases in (1) represent this scenario.

This algorithm presents several obvious issues. For one, minimum travel distance does not necessarily imply minimum travel time, which is far more conducive to minimizing wait times. For example, if an upwards-bound request is submitted on the tenth floor, and there is an idle elevator on the second floor and a upwards-bound elevator on the third, then the one on the third floor will be assigned the request. However, the upwards-bound elevator will likely need to stop along the way to offload passengers, and a stop can take a meaningful amount of time. Moreover, this algorithm optimizes travel distance in the worst-case scenario, when elevators must travel to the highest or lowest floor before switching directions, which is relatively rare.

The second algorithm I consider, *destination dispatch*, makes the wait time-optimal assignment assuming the best-case scenario. Unlike the conventional algorithm, when a passenger calls an elevator in a destination dispatch system, they enter their destination floor on a keypad outside the elevator. Hence, as soon as they make a request, the system knows not only the passenger's departure floor and direction of travel, but also their arrival floor. A destination dispatch system has more information when making a scheduling decision, so heuristically, it should be able to make a more intelligent assignment. The destination dispatch algorithm is summarized as follows:

- When an incoming request is received, initially assume that it does not get assigned to any elevator.
- For each passenger j of each elevator i, compute α_{ij}, the difference between the time at which elevator i drops passenger j off at their destination

and the current time. Essentially, execute the elevator algorithm on each elevator i until all currently assigned journeys are completed.

- Now assume that the incoming request was assigned to elevator i for each i. For each passenger j that was already assigned to elevator i, compute β_{ij}, the difference between the time at which elevator i drops passenger j off at their destination and the current time. Also compute β_i^*, the difference between the time at which elevator i drops off the passenger associated with the incoming request and the current time.

- For each elevator i, compute $w_i := \beta_i^* + \sum_j \beta_{ij} - \sum_j \alpha_{ij}$. Assign the incoming request to the elevator with minimum w_i.

Notice that when the system receives a request, destination dispatch makes the assignment which minimizes the average wait time of all service requests that have occurred thus far. That is, each assignment is optimal under the assumption that no further requests occur. Even though this assumption fails in most cases, I anticipate that destination dispatch will yield a lower expected mean waiting time of all requests made in a day. This is the performance metric that I am most interested in.

2.2 Simulation Model

The main components of my model are the building, the elevator system, and the elevator passengers.

2.2.1 Building

I consider an N-story office building consisting of one lobby at ground level, through which all building occupants enter and exit, and $N-1$ floors of offices above the lobby. I assume that the building is owned by a major corporation which occupies all floors. The lobby has an employee cafeteria as well as several restaurants and cafes outside the front entrance, incentivizing employees to return to the lobby for their lunch break. Each of the $N-1$ office floors is assumed to be functionally identical. When the corporation hosts a meeting, it is equally likely to occur on any of the $N-1$ office floors. I assume that there is a period over the night during which the building is completely vacant. While this may not be true in some major office buildings, the number of people in the building over the night is likely so few that any performance differences between scheduling algorithms are negligible during this period.

2.2.2 Elevator System

The elevator system consists of K identical elevators which service all N floors of the building. The system's scheduling policy is chosen at the beginning of a simulation run, and has no influence on the properties of the individual elevators; it only affects which requests they are assigned. All elevators begin the simulation

at the lobby. Since I assume that the building is vacant over the night, meaning that all occupants left through the lobby, these initial conditions are reasonable.

Each elevator is assumed to travel at a constant speed of $v = 30$ floors/min, based on the typical maximum speed of a passenger elevator in a mid-rise building of 300 ft/min [5] and the typical distance of 10 feet between floors in a commercial building [2]. I assume the elevator travels at constant speed because it greatly simplifies the implementation; unfortunately, however, it is not very realistic. In particular, if an elevator is between, say, floors three and four traveling upwards when it is assigned an upwards-bound request at floor four, it will promptly stop at floor four in our model. In a real system, however, the elevator would likely not be able to decelerate in time, and would need to revisit floor four later.

When an elevator stops at a floor, it remains there for a constant 10 seconds before traveling to its next request, as per a mathematical model by Johnson [3].

2.2.3 Passengers

Each elevator passenger is modeled as an individual agent in the system. Each agent has a "primary floor", representing the floor upon which their office or cubicle is located, and where they do the bulk of their work. A primary floor is assigned independently and uniformly among the non-lobby floors when the agent is created. Agents arrive in the lobby following a non-stationary Poisson process with rate function $\Lambda(t)$, described in detail in the following section. When an agent arrives in the lobby upon creation, they immediately submit a request to travel to their primary floor.

An agent submits a request to depart from their primary floor and return to the lobby after spending a normally distributed amount of time in the system, with mean $\mu_D = 4$ hours and standard deviation $\sigma_D = 1$ hour. While four hours may seem like an unusually short workday for an office employee, I clarify that an agent leaves the system whenever they return to the lobby. I assume that the vast majority of agents work for about four hours, return to the lobby for lunch, work an additional four hours to fulfill a typical 8-hour workday, then return to the lobby to leave. Although an agent is programmatically terminated when it leaves the system, the NSPP creates a influx of arrivals midway through the workday to emulate agents returning from lunch break.

After arriving on their primary floor, but before their scheduled departure time, agents may experience the need to visit a non-lobby floor. Because I assume that the building is owned by one corporation, employees may need to visit other floors for a meeting or to access facilities not available on their floor. Agents submit a request to travel to another non-lobby floor after T hours on their primary floor, where $T \sim \text{EXP}(\lambda = \frac{1}{2})$. I choose an exponential distribution

to reflect the sporadic intervals at which the need to visit another floor arises. Any given department of the corporation likely schedules meetings at regular intervals, but employees may need to meet with different departments on certain occasions, not to mention the irregular intervals at which they need to visit other floors to access facilities. The floor the agent visits is selected uniformly at random among the $N-2$ non-lobby floors that aren't their primary floor. They spend a normally distributed amount of time on that floor before submitting a request to return to their primary floor, with mean $\mu_I = 0.5$ hours and standard deviation $\sigma_I = 1$ hour. The time an agent spends on another floor depends upon how long a meeting lasts or how long it takes them to utilize facilities, both of which can be reasonably assumed to be roughly normally distributed.

2.3 Input Parameters

2.3.1 Arrival Process

The passenger arrival process is derived from data collected by Kuusinen et al. [4]. They recorded the times at which elevator passengers requested service from the lobby of a 16-story office building over the course of one workday (Fig. 1). They found that the arrival process follows a non-stationary compound Poisson process, where the *compound* qualifier reflects the tendency of passengers to arrive in batches. For simplicity, I assume that passengers only arrive individually, and model the arrival process as an NSPP. Because Kuusinen et al.'s data is from only one workday, it is inherently noisy. To decrease the noise, I compute a weighted moving average of the data to represent the arrival rate at each multiple of 15 minutes, and linearly interpolate between these points. Specifically, the arrival rate at time t, where t is a multiple of 15 minutes, is given by

$$\Lambda(t) = \frac{n_{[t-30,t-15)} + 2n_{[t-15,t)} + 2n_{[t,t+15)} + n_{[t+15,t+30)}}{6} \tag{2}$$

where $n_{[a,b)}$ is the number of arrivals recorded in $[a, b)$. Figure 2 provides a plot of $\Lambda(t)$.

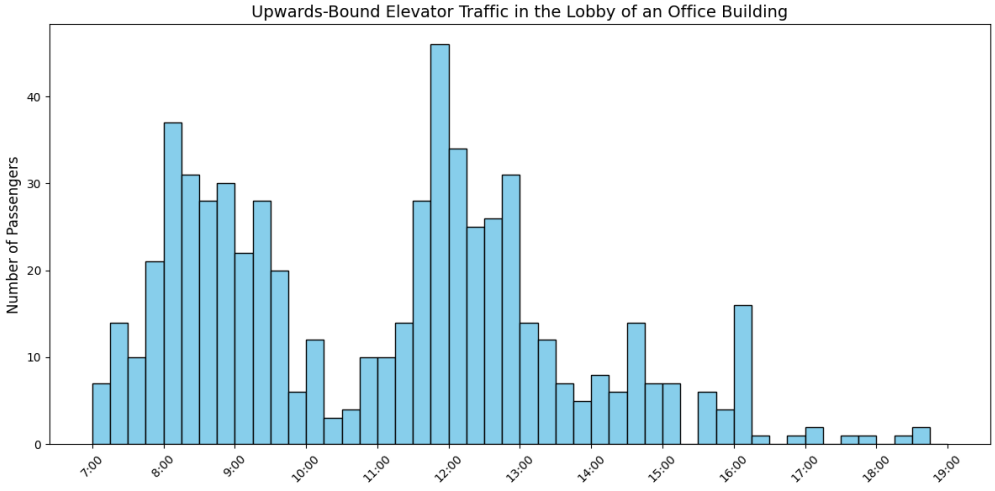


Figure 1: Histogram of times at which elevator passengers requested service from the lobby of an office building on Wednesday, October 20, 2010 [4]. No arrivals were recorded during the hours excluded from the histogram. Each histogram bucket represents fifteen minutes.

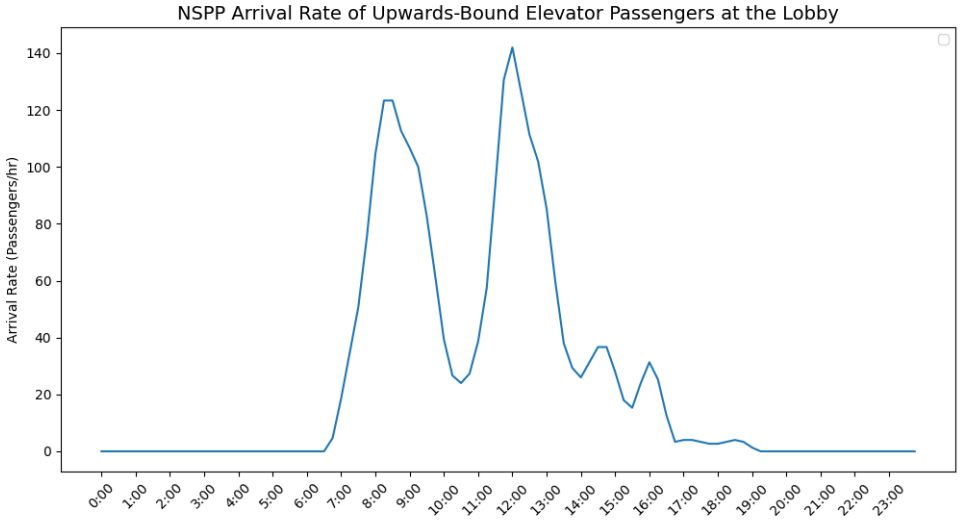


Figure 2: Plot of the non-stationary arrival rate function used for the lobby arrival process over the course of one day.

2.3.2 Parameter Summary

Parameter	Description	Value(s)
$\Lambda(t)$	Rate function for arrival process	612 arrivals over one day on expectation. See 2.3.1 for details.
N	Number of stories	10, 20, 50
K	Number of elevators	2, 3, 4, 5, 6, 7, 8, 9, 10
μ_D	Mean time an agent spends in the system before requesting to return to the lobby	4 hours
σ_D	Standard deviation of time an agent spends in the system before requesting to return to the lobby	1 hour
λ_I	Rate parameter for an agent's requests to visit non-lobby, non-primary floors	0.5 hours^{-1}
μ_I	Mean time an agent spends on a non-primary, non-lobby floor during one visit	0.5 hours
σ_I	Standard deviation of time an agent spends on a non-primary, non-lobby floor during one visit	1 hour
v	Constant speed of elevator	30 floors/min
c	Elevator stopping time	10 seconds

3 Methodology

3.1 Program Structure

All code for this project is hosted on Google Colab in a single Jupyter notebook file running Python 3.10.12. The code draws upon the paradigms of object-oriented programming and consists of five main classes, as well as several auxiliary classes which I avoid discussing for brevity. The five main classes are:

- `FEL`
- `Person`
- `FloorBatchAction`
- `Elevator`
- `Building`

3.1.1 FEL

The `FEL` class is a somewhat naive implementation of a future event list, although it is perfectly functional for my purposes. It stores a list of `Request` objects, each of which records the time of an elevator service request, the passenger's departure floor, and their desired arrival floor. The objects are sorted by their request times, and new requests are inserted via an insertion sort. `FEL` also allows for the earliest request to be peeked or dequeued.

3.1.2 Person

The `Person` class represents an individual agent in the system. It stores the agent's arrival time, departure time, and wait time for each elevator journey, among other information. All agents are generated at the beginning of the simulation and stored in a `Building` object, regardless of when they enter the system. When a `Person` is created, their initial request to travel from the lobby to their primary floor is enqueued to the FEL. They enqueue another request whenever they complete an elevator journey. That is, when they are on their primary floor, they enqueue a request to visit a different non-lobby floor. Otherwise, they request to return to their primary floor. However, if the time of their next request exceeds their departure time, they instead enqueue a request to return to the lobby.

3.1.3 FloorBatchAction

A `FloorBatchAction` object represents a group of actions that are to be simultaneously performed on a given floor by a single elevator. It stores a list of `PickUp` and `DropOff` objects, each representing a single passenger to be picked up or dropped off at that floor. All the actions share a common direction, since, for example, an elevator that is picking up an upwards-bound passenger on a floor can't also pick up a downwards-bound one.

3.1.4 Elevator

The `Elevator` class represents a single elevator in the building. It contains information such as the direction the elevator is traveling, its position, and a list of `FloorBatchActions` to be executed, sorted in accordance with the elevator algorithm. Like the building, each `Elevator` also has a clock, although it sometimes runs behind the building's clock. Each elevator functions like an independent system until a scheduling decision is made, at which point the building must know all elevators' positions at the time of the request. It is only in this case that the elevator clocks are resynchronized with the building's.

An `Elevator` object has the method `update_to_next_stop()` to update its position to its next scheduled stop, which also tabulates the wait time incurred by each of the elevator's assigned passengers between the last time the elevator was

updated and the time of the stop. It processes the `FloorBatchAction` associated with the stop, enqueueing new requests into the FEL for each passenger that de-boarded. Finally, it determines the next stop to service, and updates its direction accordingly.

Similarly, an `Elevator` has the method `update_to_time_t()` to update its position and clock to a given time t, although this method may only be used when the elevator has no stops between its current clock time and t. This method is used to synchronize each elevator's clock to the time a new request is made in order to make scheduling decisions.

An `Elevator` has the method `compute_distance()` to compute the travel distance between its current position and the floor upon which an incoming request originates, using equation 1. This is used in the conventional algorithm. Elevators also have `wait_time_diff()`, which takes an incoming request, creates two copies of the given elevator, and assigns the request to one copy. It then repeatedly calls `update_to_next_stop()` on both elevators until they've exhausted their assigned requests. Finally, it sums the wait time incurred by all passengers of each elevator, and returns the difference of the sum for the elevator with the new request and that for the one without. This method is used in destination dispatch.

Lastly, when the scheduling algorithm decides to assign an incoming request to an `Elevator`, the method `add_request()` adds the request to the elevator's `FloorBatchAction` queue. It creates a `PickUp` and `DropOff` action for the request, determines the appropriate spot in the queue to add them in accordance with the elevator algorithm, and either adds each action to a new `FloorBatchAction` or appends it to an existing one.

3.1.5 Building

A `Building` object represents the office building being simulated. It stores a list of `Elevator objects`, an `FEL`, a list of all `Persons` involved in the simulation, and the simulation clock, among other information. To run the simulation, one calls either `run_destination_dispatch()` or `run_conventional_alg()` on the object.

Both methods start by initializing the system. This involves generating all people, their arrival times, their departure times, and enqueueing their initial requests into the FEL. The simulation clock starts at time $t = 0$, which represents 06:30, the first time the arrival rate is non-zero. The simulation terminates once there are no more requests in the FEL and all elevators are idle, meaning that all agents have left the system.

If the FEL is empty but an elevator is not idle, then we call `update_to_next_stop()` on the elevator whose next stop occurs earliest, repeating this until either the

FEL is non-empty or all elevators become idle. If the FEL is non-empty, we peek the next request and determine if there is an elevator whose next stop is scheduled before the time of the request. If so, we call `update_to_next_stop()` on the elevator with the earliest stop, then peek the next request and reevaluate the same condition. This is necessary because if an elevator is dropping off passengers, it will generate new FEL requests. We can only guarantee that the FEL will not change between the present and the time of its earliest request if the elevators make no stops between the current system time and the time of the next request.

Once we've exhausted all stops before the next request `req`, we dequeue that request and call `update_to_time_t()` on all elevators, passing the time of the request. In the case of `run_conventional_alg()`, we call `compute_distance(req)` on each elevator and assign the request the one which yields the least travel distance. In the case of `run_destination_dispatch()`, we call `wait_time_diff(req)` on each elevator and assign the request to the one which yields the least increase in wait time. We now check again if the FEL is empty, repeating until the termination condition is met.

3.2 Setup and Data Collection

3.2.1 Running the Simulation

To run the simulation, we start by initializing two `Building` objects with the desired number of floors and elevators. After this, we reset the random seed to a random value using `random.seed()`, and call `run_conventional_alg()` on one of the buildings. Once the simulation is complete, we again update the random seed to a new random seed as before, and call `run_destination_dispatch()` on the other building.

To access the wait time data, I first create two empty lists per performance metric, one for each algorithm. Each instance of the `Person` class stores all the wait time information for each of the person's elevator journeys. Thus, to collect this data I loop over each building's list of people and append the times to their corresponding lists.

3.2.2 Data Collection

To help consolidate the data, I use the Python packages `numpy` and `pandas`. To compute a performance statistic from a list of wait times associated with a simulation run, I use `numpy.mean()` or `numpy.max()` and record this value to a pandas `DataFrame` at position (run, "(algorithm)_(metric)") where "run" is the current replication number.

I loop over this entire process to collect sufficient data, with the results from each iteration being saved to the dataframe.

3.2.3 Summary Data

To find the overall mean values across multiple replications, the `mean()` function is called on the dataframe. This provides the mean value for each column of the dataframe, which, in this case, is each performance metric. Similarly, to find the variance of this data, I use the `var()` function of the dataframe.

All confidence intervals are calculated using a custom Python function. This function makes use of the `t` object from the library `scipy.stats`. Given an alpha value and the degrees of freedom, the t distribution is sampled using the function `t.ppf((alpha)/2, df=(degrees of freedom))`.

4 Results and Analysis

4.1 Fixed Parameter Analysis

First, I analyze the case with 2 elevators and 10 floors. In Table 1, I compare the mean and maximum waiting times between the conventional elevator algorithm (C.A.) and the destination dispatch algorithm (D.D.) across 30 replications. I then use this data to calculate the mean value of each category of data collected, as well as the variance and 95% confidence intervals (C.I.).

Table 1: Waiting Times (Minutes) - 2 Elevator - 10 Floors

Run	C.A. Mean	D.D. Mean	C.A. Max	D.D. Max
1	0.844347	0.792197	4.112635	2.822669
2	0.844353	0.801537	3.261925	3.254562
3	0.842136	0.834475	3.459567	3.541855
4	0.859863	0.791575	3.729820	2.958461
5	0.867600	0.793759	3.996822	3.428894
6	0.829520	0.771752	3.281672	2.792802
7	0.872927	0.753831	3.525960	2.559251
8	0.864313	0.758353	3.883203	2.622724
9	0.885424	0.817031	3.890078	3.434089
10	0.844123	0.801174	3.611790	2.988077
...	...	...	...	...
30	0.852545	0.757260	3.459408	2.674117
Mean:	0.848892	0.791335	3.617253	2.953242
Var:	0.000611	0.000728	0.087733	0.062486
C.I. +/-	0.009229	0.010075	0.110602	0.093341

Now I analyze the difference between the algorithms for each metric.

Table 2: Comparison of Mean and Max waiting times (C.A. - D.D.)

(a) Minutes

	Mean	Max
Difference	0.057556	0.664011
C.I. +/-	0.010668	0.109048

(b) Seconds

	Mean	Max
Difference	3.45336	39.84066
C.I. +/-	0.64008	6.54288

As both of these metrics have an associated 95% confidence interval which does not contain 0, the results are statistically significant. This indicates that the destination dispatch algorithm is better than the conventional algorithm in both mean waiting time and maximum waiting time. The mean waiting time in the building with the destination dispatch algorithm is approximately 3.45 seconds lower on expectation, and the maximum waiting time is approximately 39.84 seconds lower.

Interestingly, the conventional algorithm has a lower average queue waiting time (time a passenger waits for an elevator to pick them up) in this scenario.

Table 3: Analysis of Queue and Travel Times (Minutes)

(a) Queue Times

Run	C.A. Mean	D.D. Mean
1	0.392241	0.393069
2	0.402848	0.444230
3	0.407306	0.414112
4	0.409544	0.391392
5	0.385613	0.425502
...	...	...
40	0.371757	0.394865
Mean:	0.398505	0.409412
Var:	0.000294	0.000279

(b) Travel Times

Run	C.A. Mean	D.D. Mean
1	0.456682	0.374085
2	0.450736	0.389945
3	0.461066	0.383843
4	0.452843	0.384297
5	0.443736	0.394933
...	...	...
40	0.439338	0.381959
Mean:	0.452073	0.385063
Var:	0.000108	0.000044

Again, I take the differences between the results of each algorithm in each metric to view their relative performance.

Table 4: Comparison of Queue and Travel waiting times (C.A. - D.D.)

(a) Minutes

	Queue	Travel
Difference	-0.010907	0.067010
C.I. +/-	0.007537	0.003895

(b) Seconds

	Queue	Travel
Difference	-0.65442	4.0206
C.I. +/-	0.45222	0.2337

Note the number of replications was increased to 40 in order to achieve statis-

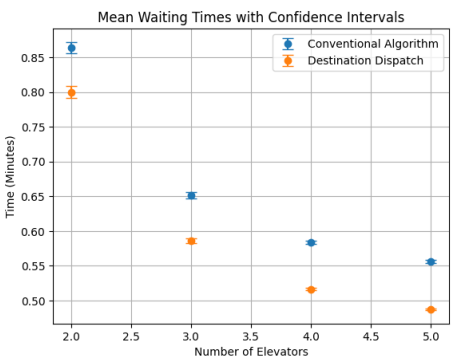
tically significant results in both metrics.

This shows that the conventional algorithm performed better than the destination dispatch algorithm in average queue waiting time by approximately 0.65 seconds. Hence, the destination dispatch algorithm achieves its lead in overall wait time from its significantly lower travel times. On average, the destination dispatch algorithm is approximately 4 seconds faster in transporting passengers from their origin floor to their destination.

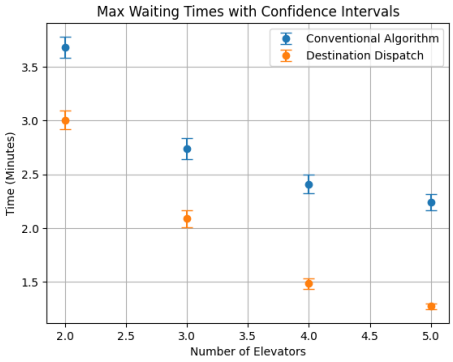
4.2 Varying Number of Elevators

The destination dispatch algorithm continues to beat the conventional algorithm as I vary the number of elevators in the building.

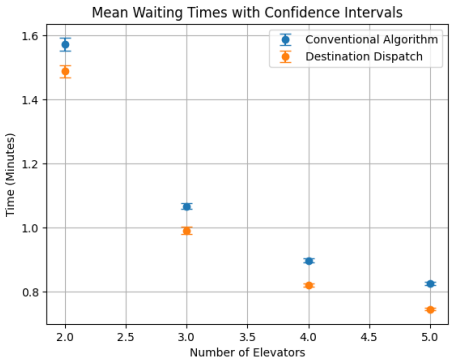
First I analyze two buildings: one with 10 floors, and one with 20 floors. In each building, I run the simulation with 2 to 5 elevators and record the results. All confidence intervals are 95% confidence intervals and each data point is the mean across 40 replications.



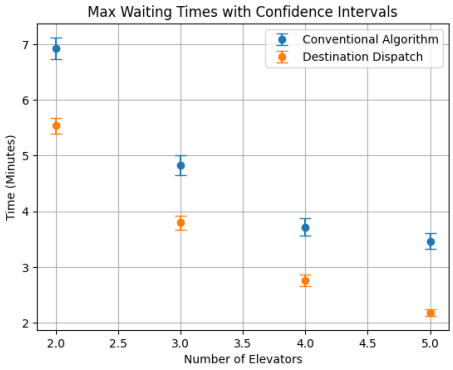
(a) Mean Waiting Time - 10 Floors



(b) Max Waiting Time - 10 Floors



(c) Mean Waiting Time - 20 Floors



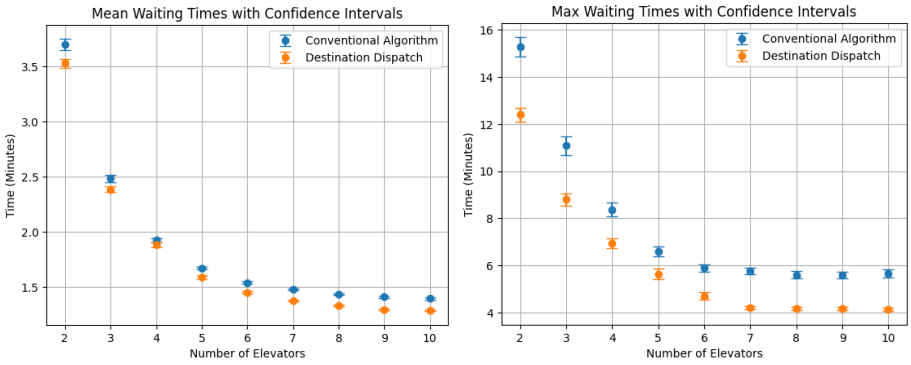
(d) Max Waiting Time - 20 Floors

Figure 3: Comparing results when varying the number of elevators from 2 to 5.

In each of these graphs, we see that the destination dispatch algorithm outperforms the conventional algorithm by a statistically significant margin. As the number of elevators increases, we see that in both the 10 and 20 floor cases the destination dispatch algorithm manages to scale slightly better than the conventional algorithm, increasing the difference between the mean waiting times.

However, the most notable impacts of the destination dispatch algorithm can be seen in the maximum waiting times. As the number of elevators increases, the maximum waiting time decreases substantially faster for the destination dispatch algorithm compared to the conventional algorithm. This effect is strong enough that in some cases the destination dispatch algorithm performs better than the conventional algorithm with an extra elevator. This can be seen when the destination dispatch algorithm has 4 elevators and the conventional algorithm has 5 in all of the graphs.

To perform a stress test, we can increase the number of floors to 50 and vary the number of elevators from 2 up to 10.



(a) Mean Waiting Time - 50 Floors (b) Max Waiting Time - 50 Floors

Figure 4: Varying the number of elevators from 2 to 10 in a large building.

Increasing the range in which we compare the number of elevators allows us to see the point of diminishing returns as we add more elevators. However, we see that even once this point is reached and neither algorithm has any significant performance gains, which seems to be around 8 elevators, the destination dispatch algorithm still outperforms the conventional algorithm.

5 Conclusion

The results align with my predictions precisely: for every choice of N and K that I considered, the destination dispatch algorithm yields shorter expected mean and maximum wait times than the conventional algorithm by a statistically significant margin. The margin, however, is quite small in most cases. At this point, we must consider practical significance instead of statistical significance.

In terms of elevator passengers' satisfaction, an expected difference in wait time of only a few seconds has a negligible impact. While the difference in maximum wait time is often on the order of minutes, which would affect the satisfaction of individual passengers, this improvement only benefits a few elevator journeys among the thousands that occur each day. If we wish to upgrade an existing building's elevator system, and we aim to optimize customer satisfaction, then installing destination dispatch elevators is unlikely to be worth the cost. For a luxury hotel, for example, customer satisfaction is paramount to their success as a business. Assuming that my results hold true for the elevator traffic patterns of a hotel, upgrading to a destination dispatch system is likely financially unproductive. Indeed, it may even be counterproductive, since my simulations fail to factor in certain aspects of human behavior. Most guests will be unfamiliar with destination dispatch systems given their scarcity, and will face a learning curve in operating one. The confusion of entering your destination floor instead of merely indicating your direction of travel could very well take more time to resolve than would be saved by the destination dispatch algorithm.

However, in our running example of an office building owned by a major corporation that is considering upgrading their elevator system, a difference in average wait time of only a few seconds may actually be practically significant. In this case, the corporation's concern is less so passenger satisfaction and more so passenger productivity. And since the same passengers use the elevators day after day, delays resulting from confusion in operating a destination dispatch system would quickly be eliminated. Considering the case of ten floors and two elevators, and given that about 3000 elevator journeys occur per day in my model, a rough-cut calculation reveals that the destination dispatch algorithm saves approximately

$$\frac{3.45 \text{ sec}}{\text{journey}} \times \frac{3000 \text{ journeys}}{\text{day}} \times \frac{260 \text{ workdays}}{\text{year}} \times \frac{1 \text{ hour}}{3600 \text{ sec}} = \frac{747.5 \text{ hours}}{\text{year}} \tag{3}$$

If we value a typical passenger's time at \$40/hour, we gain approximately \$29900 in revenue annually by upgrading to a destination dispatch system. While this analysis is obviously highly approximate, it suggests that it is possible for even marginally shorter elevator wait times to offset the costs of upgrading a conventional elevator system to use destination dispatch.

Thus far, I assumed that we are upgrading an existing elevator system. How-

ever, if we are planning the construction of a new development and are concerned with the installation and maintenance costs of the elevator system, then it may be profitable to consider destination dispatch. While the benefit depends heavily upon the number of floors and the intensity of the traffic, my results reveal that in some cases, destination dispatch outperforms the conventional algorithm with an extra elevator. This effect becomes apparent when we consider systems with more than two or three elevators. If we are planning the construction of a skyscraper with a dozen elevators, we may use destination dispatch instead of conventional scheduling and opt for one fewer elevator, all without sacrificing passenger wait time. While installation costs for conventional versus destination dispatch systems are not readily available, the cost of installing and maintaining an additional elevator is likely far greater than the cost of upgrading to destination dispatch.

The assumptions involved in constructing this model are not accurate in all real world scenarios. For example, to reduce complexity, I assumed that all elevators travel at a constant speed, neglecting acceleration and deceleration. The time needed for an elevator to accelerate and decelerate is not negligible, and may meaningfully affect the accuracy of my data. The assumptions made about the purpose of the building significantly impact inter-floor travel. In this model, I assumed that one company owns the entire building; thus, it is plausible that passengers may need to travel between any two floors. This is the case for some office buildings, but many are shared among multiple companies. It is unlikely that passengers would travel between floors owned by two different companies.

Additionally, the arrival process used in this model is relatively light. A building with higher arrival rates at its peak times may yield a stronger disparity between the two algorithms. This is apparent in real-world results, such as the New York Marriott Marquis Hotel, with 49 floors and up to 12,000 people in it at any given time [1]. Since upgrading their elevators with Schindler's Destination Dispatch system, delays were reduced by more than 50%. Besides higher arrival rates, the input model also does not account for people arriving in groups. Often times people will arrive to work in groups due to carpooling or public transit. If these people travel to different floors, their wait time would increase in comparison to if they arrived several minutes apart. Another issue with the model is that the capacity of each elevator is not enforced. In a real world scenario, capacity constraints could cause people to have to wait for another elevator, drastically increasing their wait time. As my input model only allows for individual arrivals, though, capacity issues are far less likely than if group arrivals were considered.

References

[1] New york marriott marquis hotel. https://group.schindler.com/en/media/references/new-york-marriott-marquis-hotel.html.

[2] Bumseok Chun and Jean-Michel Guldmann. Two- and three-dimensional urban core determinants of the urban heat island: A statistical approach. *Journal of Environmental Science and Engineering*, B1:363–378, 01 2012.

[3] B. M. Johnson. User requirements of elevators. *Canadian Building Digest*, Oct 1977.

[4] J-M Kuusinen, J Sorsa, M-L Siikonen, and H Ehtamo. A study on the arrival process of lift passengers in a multi-storey office building. *Building Services Engineering Research and Technology*, 33(4):437–449, Nov 2011.

[5] Javier Vazquez-Esparragoza and Giovanni Puccini. Logistics of personnel movement by elevator. https://www.digitalrefining.com/article/1002246/logistics-of-personnel-movement-by-elevator, 2018.