Elevator Simulation

CS166

Anna Pauxberger, Michael Yang, Ian van Buskirk, Rita Kurban - 16 January 2018

Introduction

The purpose of this paper is to introduce a simulation of a building containing a single elevator and some number of passengers who want to travel from one floor in the building to another. Anna, Michael, Ian, and I collaborated on two distinct strategies and compared their efficiency by plotting the results. We used object-oriented programming to implement three classes: elevator, building, and passenger and combined them in the main body of the simulation.

Strategies

In our simulation, we implemented two strategies where one is a significant improvement over the other. We decided to keep them simple for the learning purposes and make it possible for everyone to contribute.

Strategy 1: Basic Controller

The first strategy is not the most efficient one, but it is very easy to understand and implement. The elevator goes all the way up or down, stops at every single floor, and lets people in and off if there is a need. It changes its direction once it gets to the first or the last floor and repeats the cycle until all the passengers arrived at their destinations.

The output of this function is the `elevator` and the `building` where it is situated, the `timestep` which we later use in the efficiency calculations as well as a Boolean variable `is_moving_up` which states whether the elevator moves up or down. The timestep is being increased each time the elevator makes a stop. The output of this function is the timestep as well as the direction in which the elevator is moving.

Strategy 2: Secondary Controller

The second strategy is similar to the first one. However, the elevator only stops if there is a need. Each time, the program checks if there are people on the floor with the help of the `building.waiting_passengers` attribute which is a dictionary that keeps track of the passengers who want to get in on every single floor. The second dictionary is `elevator.passengers`, and it stores the instances of the `passenger` class who want to get off the elevator on a specific floor. As a result, the elevator only opens the doors in case passengers need it which makes this strategy more efficient. Another important feature of this strategy is that it accounts for the elevator capacity what the first strategy

failed to do. The elevator won't stop to let new passengers in case it is full which makes this strategy more realistic as well.

Efficiency Measurements

In the main body of the program, we coded a few helper functions which allow us to measure the efficiency of the strategies based on their running times. The first two functions: `display_state` and `run_simulation` are designed to run the simulation until all the passengers are processed and display the number of passengers inside the elevator and on the floors on each step. It helps us make sure that the simulation is running smoothly and there are no bugs in the code.

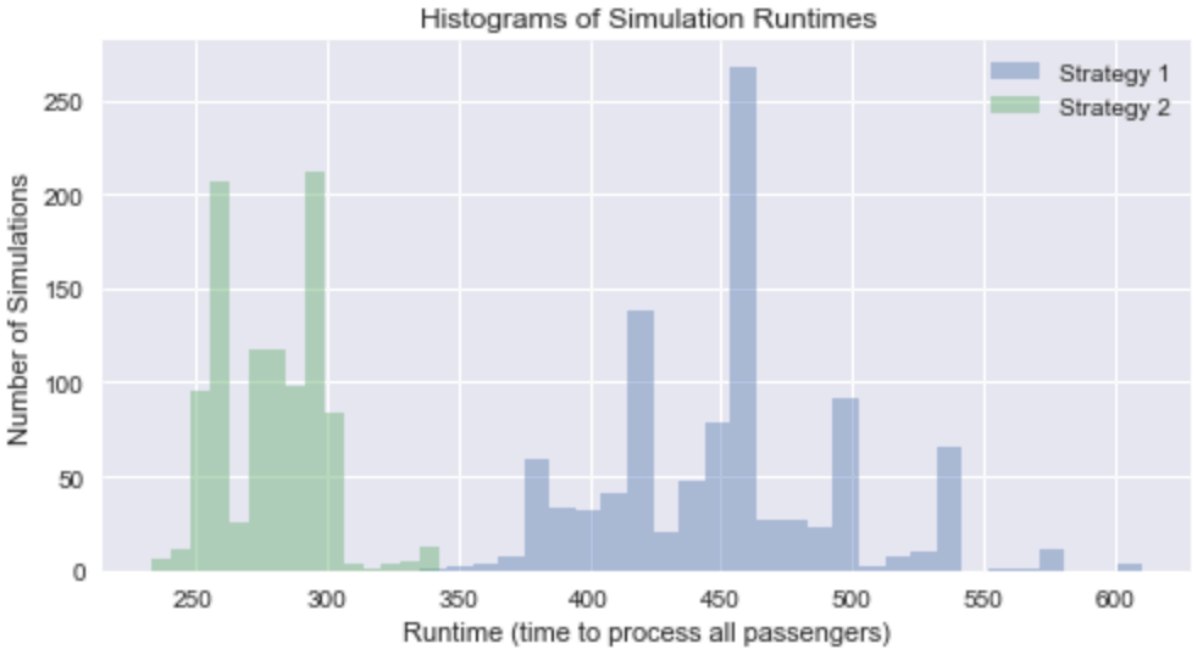
After that, we created two identical `elevator` and `building` objects to make sure that no external factors (such as a different number of people or floors) bias our results. We then compared two strategies by running them `runs` times inside the `compare_strategies` function which uses the `extract_times` helper to return metrics (`"runtimes","waits","travels","totals"`) that we used for further analysis.

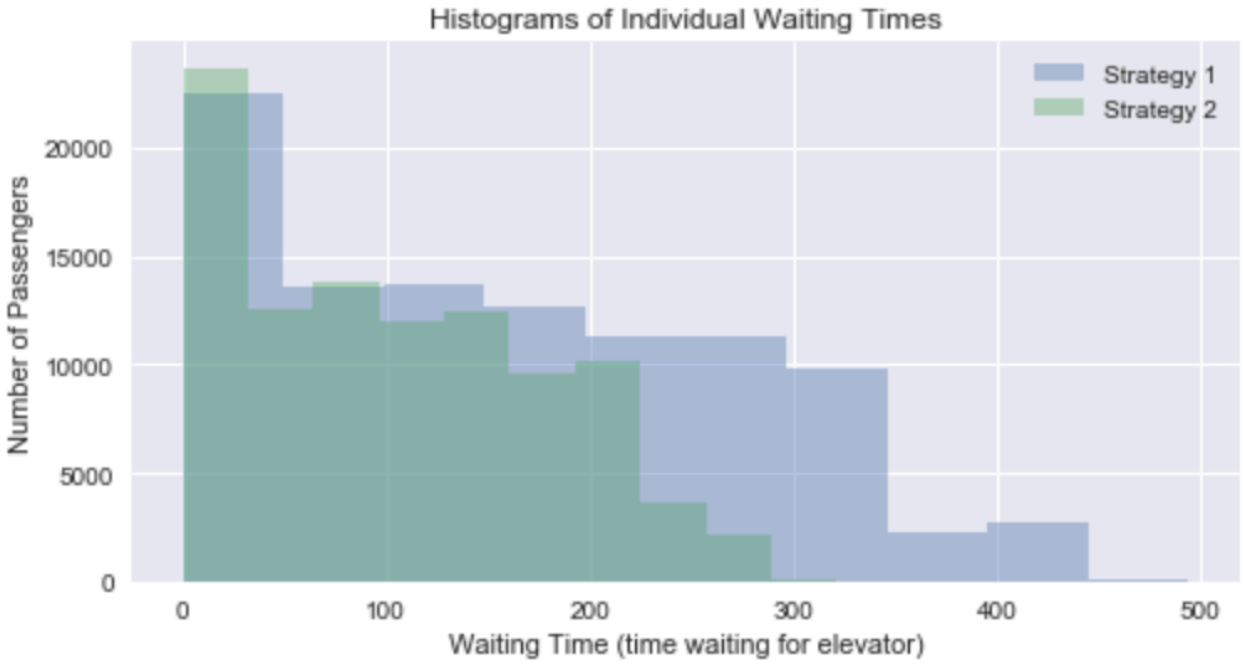
We first presented the running times in a tabular form to give a general understanding of which strategy performs better. As expected, the running times were much lower for the second strategy.

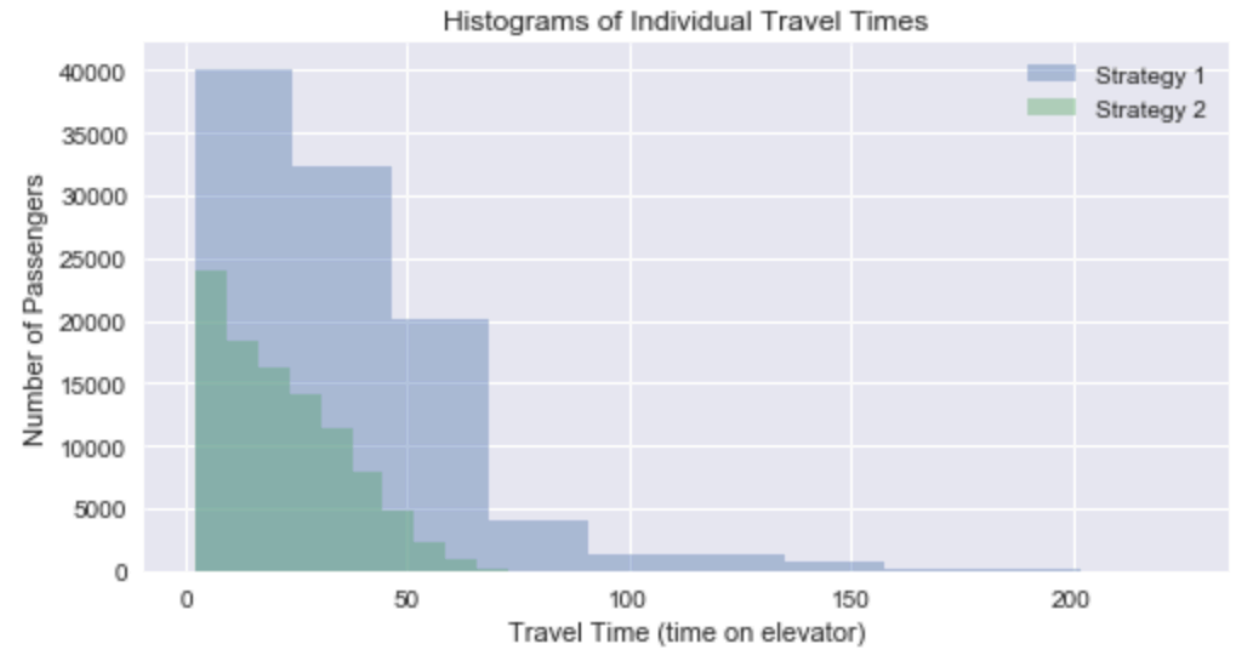
	Strategy 1	Strategy 2
count	1000.000000	1000.000000
mean	451.708000	279.009000
std	43.827521	19.241808
min	336.000000	234.000000
25%	420.000000	259.000000
50%	456.000000	279.000000
75%	470.000000	296.000000
max	610.000000	343.000000

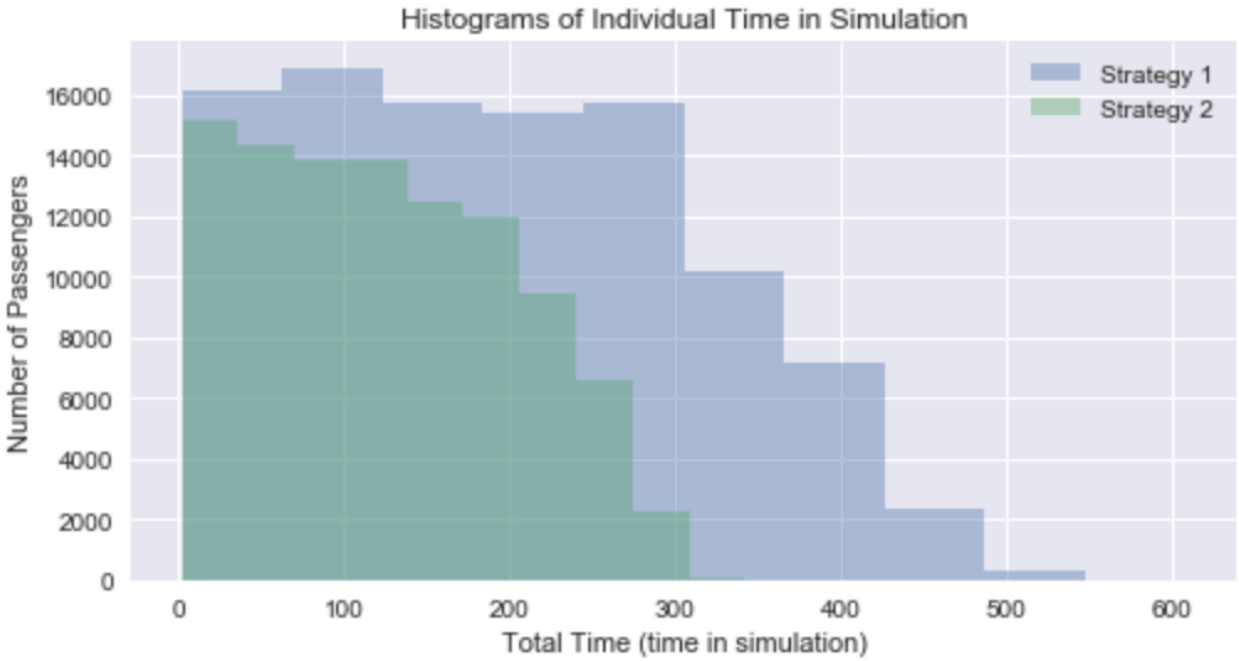
Table 1. A summary of the simulation runtimes under each strategy

To further analyze the efficiency and present the data in a readable format, we generated a few histograms that present the runtimes, waiting times, travel times, and individual travel times for both strategies.









To create these bright illustrations, Ian proposed to use the `seaboard` statistical data visualization package. He was also the person who was responsible for this chunk of work. I never used this package before, and it was a nice learning experience to talk to Ian *(who kindly explained everything)* and recreate similar plots on my own.

Results

Our efficiency measurements illustrate that the second strategy outperforms the first one in a vast majority of cases. The second strategy is also more realistic as it accounts for the elevator's capacity. However, both of them are not ideal for a real-life scenario. If I were to implement a different strategy, I would make it possible for people to state their direction prior to the boarding so that they don't have to go all the way down in case they need to go up or vice versa which would also reduce the number of unnecessary stops and optimize the use of the capacity. Thanks to the object-oriented programming, it would be an easy thing to do, as there is no need to rewrite the entire code as I can reuse certain parts of the code, in particular, classes.

Contributions and Learning Outcomes

Even though I'm not new to programming in Python, I never used object-oriented programming before and had a vague understanding of how classes work. A big chunk of my learning experience was to discuss the logic of the simulation with my peers and understand how generic classes can be combined with each other and used to create a working simulation. Michael was very helpful in explaining some of the general concepts and strategizing. My major contributions were writing up the classes, coming up with attributes and methods, and structuring/editing the code to make it clear for Anna as well as myself. We worked on classes together with Anna and Ian, while Michael made some improvements upon our work as he joined a little bit later. I also learned what doc strings are and wrote the documentation for the first two classes as well as for some functions from the main body of the simulation. Even though the strategies were originally written by Michael and Ian, Anna and I worked independently to make sure we can recreate them on our own and made some minor improvements to make the code more concise and clear.

To sum up, Anna and I mostly worked on the classes, their methods and attributes as well as documentation. Michael was responsible for the main body and guided us through the entire process which I'm extremely thankful for. And Ian created the illustrations and contributed to other parts of the simulation greatly. We all worked together to plan the structure of the simulation, understand the benefits of object-oriented programming and why it's suitable for this assignment, and proofread the code once it was ready. As a result, I am confident about using classes and feel excited to implement the third strategy to make sure I'm not only able to reproduce the code I've seen before but also to write a novel strategy on my own.