ISAAC SCHAAL
CS 166: Elevator Simulation

with JiaHer Teoh and Stanislav Chobanav

The python code for this is located in the supplementary document or on GitHub at https://gist.github.com/isaacschaal/ccfbb2fef4caf96de19d33ddcf4ad246

In this report, I summarize the construction of and results from an elevator simulation designed to test the efficiency of various elevator strategies. We created `Elevator`, `Building`, and `Passenger` classes in python to simulate the agents in the system, and included multiple elevator strategies in the methods within the Elevator class. In order to compare the strategies, we produced a video showing the behaviour of the elevators and we rigorously tested and compared the strategies under a diverse set of conditions.

Strategies:

(This is a high level overview of the strategies. For a more technical explanation and the implementation, see the commented code in the supplementary document)

We were presented with an initial elevator strategy and tasked with devising a new strategy and comparing the two. The first strategy was for the elevator to start on the bottom floor, move up to the top floor, return to the bottom floor, etc. while picking up passengers on the way. In our interpretation of this, the elevator only picked up people whose destination was in the direction of travel of the elevator and the elevator only stopped on a floor if someone was going to get on. We implemented this in the `simulation_2` method of the `Elevator` class.

We then created our own algorithm. The strategy is that the elevator starts at the bottom, and begins to move up. It continues going up as long as there are passengers inside the elevator who want to go up or there is someone waiting for the elevator on a higher floor. As it is moving, it only lets on passengers whose destination is in the direction of travel. For example, as the elevator gets to the third floor from the second, it will let on a passenger who is going to the fourth floor, but will wait until it is heading down to let on a passenger who is going to the first floor. Once the elevator has dropped off all passenger and no one is waiting for the elevator above it, it switches direction and begins to move down. It picks up all passengers who want to go down, and continues going down as long as someone wants the elevator to go down. It then repeats the process.

This strategy can be thought of as a modification of the first strategy. The elevator will continue in its direction of travel as long as there is a request (either a destination of a passenger or someone waiting to get picked up) in that direction. When there are no longer requests in that direction, it switches directions and repeats the process. This manifests by the elevator moving to the most extreme request in its direction of travel. If there were always people on the top and bottom floors, the two strategies would play out the same. However, our strategy is hopefully more efficient, as it cuts out wasted travel in a direction when it is not needed. The implementation of this strategy is in the `simulation` method of the `Elevator` class.

Efficiency Test:

Once we implemented the strategies, we needed to run a simulation and compare the two strategies. We first created the infrastructure for the simulation to run. In order to incorporate randomness into our simulation, we sampled from several distributions to create the passengers and building. Firstly, there was the issue of the delay between passengers. We created a Poisson distribution with a variable mean that is sampled from in order to get the delay times. This ensured that the delays were random enough to reflect real life while also centered around a changeable value. We also wanted to ensure that the delay time could be zero, so the Poisson distribution was shifted to the left so that outputting negative values was possible, and all negative values were treated as delay of zero. We also used a modifiable distribution for where the passengers start at and where they end at, with each floor being able to be given a different comparative likelihood of being chosen. The next step was to span the passengers, initialize the elevator, and run a loop through the main command (`simulation` or `simulation_2`) that runs while there are still passengers.

We first created a video that helps to show how both strategies operate and verify that they are successful. This can be seen at https://www.youtube.com/watch?v=dXrSLr90-s8

However, one run of each strategy is not enough to compare on. This video is an illustration that shows the workings of the algorithms in practice. However, it does not tell us much about the efficiency of the algorithms. The individual passenger experience is what the algorithm is trying to optimize, and this is made up of several different components. Similarly, just running the simulation once leaves the results up to chance. Accordingly, we devised a rigorous test to compare the two strategies on different metrics in a range of different scenarios over multiple iterations.

The prerequisite for both of these strategies is that they work correctly. It doesn't matter how fast the elevator moves people if it brings them to the wrong place. In all instances, both algorithms work correctly, bringing people to the right places. Once this is verified, we can compare the efficiency. We used several metrics to test the efficiency of the strategies. The main goal is minimizing wait time for passengers. The two strategies run on the same building and passengers, which moves at the same speed and takes a constant time to let people in and out of the elevator. Thus, comparing several aspects of the wait time is an appropriate metric of efficiency.

We chose the average, 95th percentile, maximum and median wait time over the course of the simulation (which has 200 total passengers). We did this for a “typical” case, where the specified conditions equate to reality (delay time is appropriate, first floor is more likely to be destination and origin, 12 floors). We also wanted to test the two strategies under more extreme conditions. We again compared the two strategies using a very crowded building (low delay time), a very tall building (23 floors) and a building where some non-ground floors are very popular.

Results:

(The full description of the results from the simulations can be found in the supplementary document. This section will present the most important ones, with explanations)

The strategy that we designed `(simulation)` was superior to the initial strategy `(simulation_2)`. In the test approximating normal conditions, our strategy gave an average time from pressing the button to getting to the destination of just over 29 seconds, while the average time for the alternative strategy was almost 45 seconds. This represents a significant increase in efficiency. The median, and 95th percentiles were also lower for the our strategy, while the maximum wait time was similar but slightly better for our strategy.

The wait time from pressing the button to getting in the elevator was much lower for our strategy, while the time in the elevator was quite similar between the two. These results make sense because the main efficiency gain from our algorithm comes when the alternative algorithm continues to the top or bottom of the building unnecessarily while ours avoids this.

We also wanted to assess our algorithm under diverse conditions. Our algorithm performed well in comparison to the alternative in the tall building scenario, which makes sense as the alternative algorithm’s inefficiency is exaggerated by tall buildings. It also performed well in the skewed scenario. However, the efficiency increase of our algorithm was much lower in the crowded scenario, when many people were getting on the elevator (the delay time was low). This result also makes sense. When there are many people waiting for the elevator, there is more likely to be passengers waiting on the top and bottom floor. When there are, our algorithm acts just the same as the alternative. In fact, the median time for getting to destination for our algorithm was slightly higher, at 88 seconds, than the alternative of 86.5. However, for the other metrics our algorithm is still slightly superior. Overall, from this testing of extreme scenarios, our algorithm performs at best much better and at worst the same.

In our discussion of the results, we can also examine possible scenarios that might affect our model. One such example is that if the original algorithm was implemented `(simulation_2)` on an especially large building, people might learn the schedule of when the elevator comes (as it is infrequent) and build their schedules around this. This could potentially make the algorithm perform much better. This idea showcases that similarity between our model and other real life scenarios. With very few modifications, our code could simulate a subway, where the traditional algorithm is to go from one end to the other. If we changed the passenger size, time, and the ability to push buttons, we could easily implement this “subway model” and the widely accepted preferable algorithm is the exact algorithm that we improved upon in this simulation. This example highlights the importance of the real life scenario that is being modeled, as well as the similarity between different models and algorithms.

Contributions

As a team we brainstormed how the simulation would work overall and what our algorithm should be. I then started building the elevator class, with the required attributes and methods. Several of my main contributions were implementing the idea of our elevator algorithm into python along with (the much simpler task) of translating the original algorithm as a comparison. I also designed and implemented the comparative simulation, writing the loop that built the lists of results and creating the function to display them in a meaningful way. We also worked as a group debugging everything, commenting, making it look nice, and adding features in order to get to the final product.

Python Implementation

During the process of this assignment, I learned a lot about how to implement a simulation in python. I learned several more efficient way to implement something in the python language, but the bulk of what I learned was in the mindset and strategies for creating a simulation from scratch. Looking at the code now, there are hundreds of lines that form a complicated procedure. However, we started out with a few lines defining an elevator. I learned how to break up a big idea into smaller parts and progressively implement each part one at a time. I also gained experience making sure that all of the individual parts work together seamlessly and knowing what other things need to be changed when one aspect is changed. I practiced clearly stating what I want to have happen, and then thinking about how to make that happen in code. We also spent time anticipating and observing edge cases and determining the right way for them to be handled.