Elevator Simulation

Authors

- Zineb Salimi
- Michael Chen
- Berfin Karaman
- Khoi Pham

Background

Simulation for an elevator is built for the purpose of testing the different configurations needed to meet a projected demand from passengers. For example, a large elevator might never overload with passengers but it would incur a great cost on the building's management. In contrast, a too small elevator could be full most of the time, significantly increasing the waiting time of each passenger. Therefore, performing experiments in silico for different sets of passengers and their floor requests on elevators of determined configurations can help us design a suitable elevator.

About the simulation

In this assignment, our group has implemented a simulation of a real life elevator which consists of three main classes:

- `Passenger` describes a single passenger and their properties: which floor they are going to, whether they are still waiting or already in the elevetor, and their unique ID, and even the time they take to enter and exit an elevator.
- `Elevator` is a parent class which descibes the movement of elevator from floor to floor, the time it has taken to perform operations (move to one floor, open and close doors).
    - Its daugther classes, `ElevatorSimple` and `ElevatorAlgo1` define the algorithms (strategies) to find the next floor to go to, either to pick up or drop off a passenger.
- `Building` class manages passengers and elevator: it can generate a random scenario of unique passengers with different floor request and run elevator on such a request.
    - Upon the calling of `simulate()`, this class manages the states of the passengers and updates the elevator's destinations list.

There are a few main parameters the user can input to the simulation via class `Building`:

- This class can be initialized with `bottom_floor` and `top_floor` which indicate the building's floors. By default, for bottom floor, a random integer will be generated from $[-5; 0]$; for top floor, an integer will be from $[0; 100]$.
- Method `generate_passengers()` can be parametrized with `num_pass` which indicates the total number of passengers we would like the scenario to include.

- Method `simulate` allows us to specify the following parameters:
    - `elev_algo`: elevator strategy to be used, can be `'simple'` or `'demand'`
    - `capacity`: specify elevator's capacity

Elevator strategies

We have implemented one strategy from the prompt and one based on local optimum heuristic. In the first strategy, the evelator starts at the bottom floor and goes up until it reaches the top and then goes down; this pattern continues until all passengers have arrived. At each floor, this simple strategy opens and closes doors, even if no passenger is on that floor or will exit at that floor.

The second strategy is called "on-demand" and uses the local optimum heuristic. The idea is that, capacity permitting, we would like to go to the next closest floor. This will process the requests that are the least time-consuming first, which should, in theory, decrease the average waiting time (this is analogous to letting people in the groceries queue with only a few items to go first). If the capacity is not full, this elevator will go to the closest destination of passengers in the elevator **or** the closest request floor of passengers still waiting. If the capacity is full, it will only go to the closest destination.

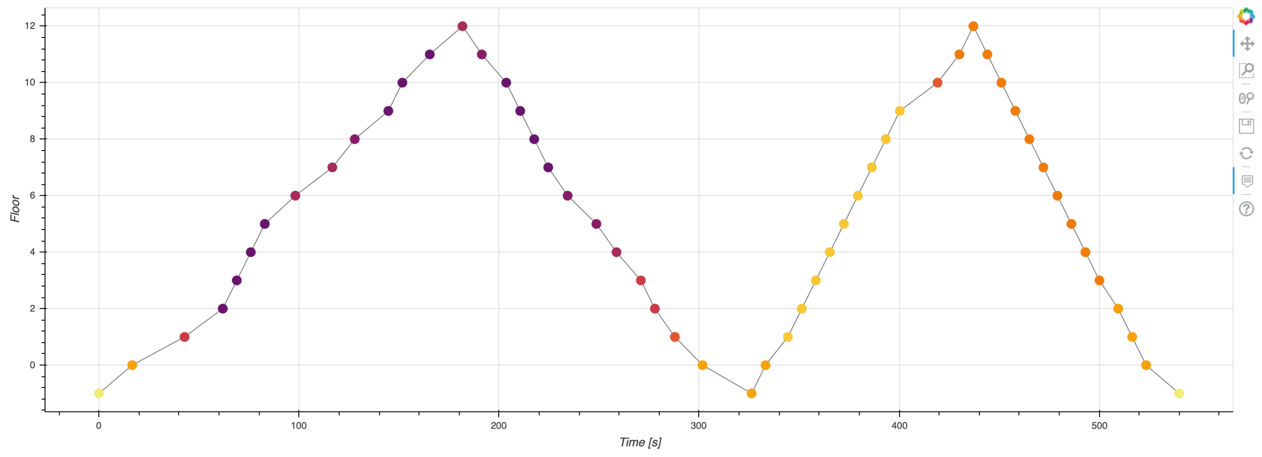
Efficiency comparison

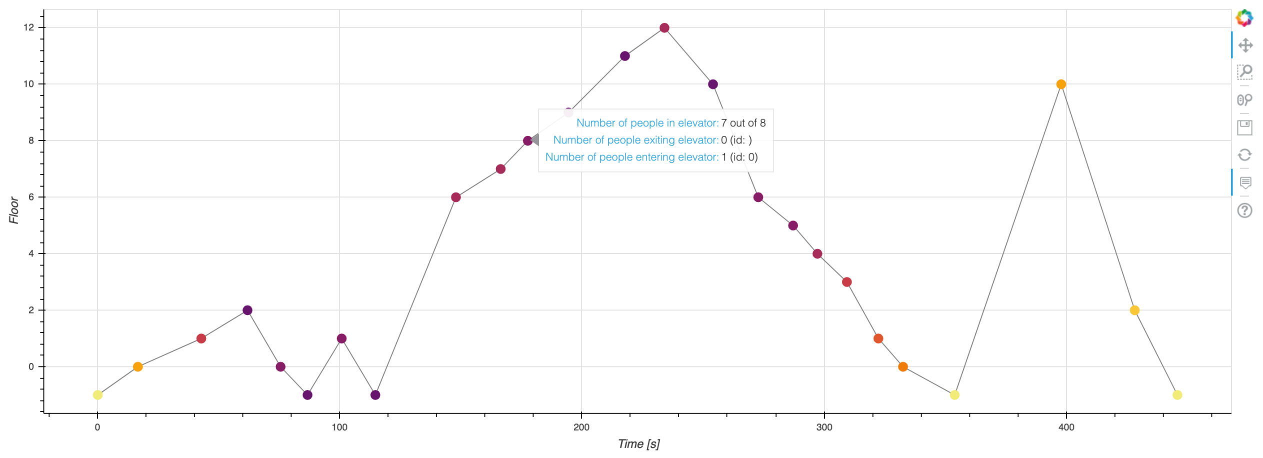
To compare the two different strategies, we employ two metrics:

- Time taken by the elevator to complete a sequence of operations
- Average waiting time of each passenger

The reason we use these metrics is to observe the overall operation time of an elevator (e.g., useful for analysis of electricity cost) as well as each passenger's experience — was the average passenger affected if the total time an elevator spent decreased.

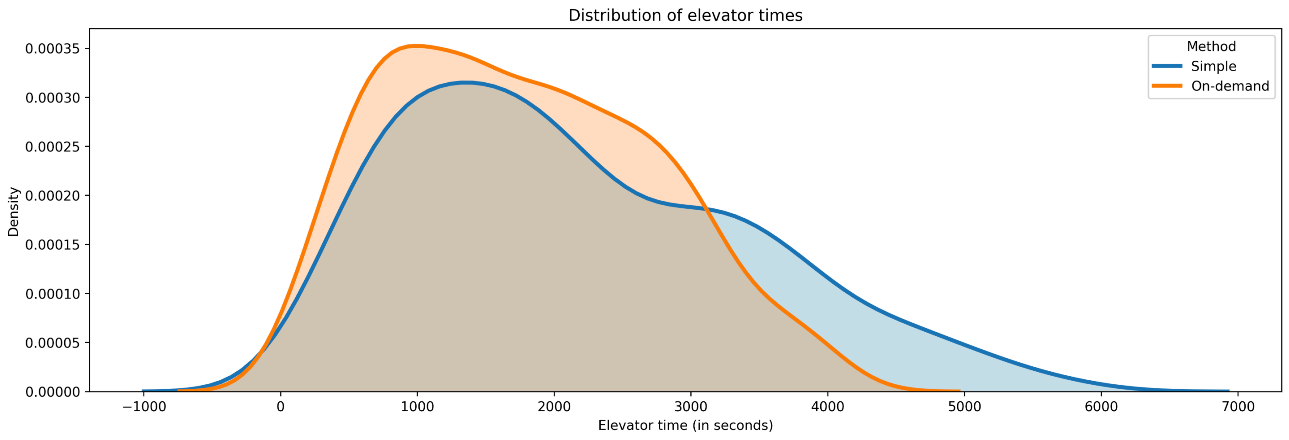
We decided to present the results on a granular level (for one simulation run) as well as an aggregate analysis of the employed metrics. Here is an example of the granular level comparison between the two algorithms (top figure is "simple", bottom is "on-demand"):

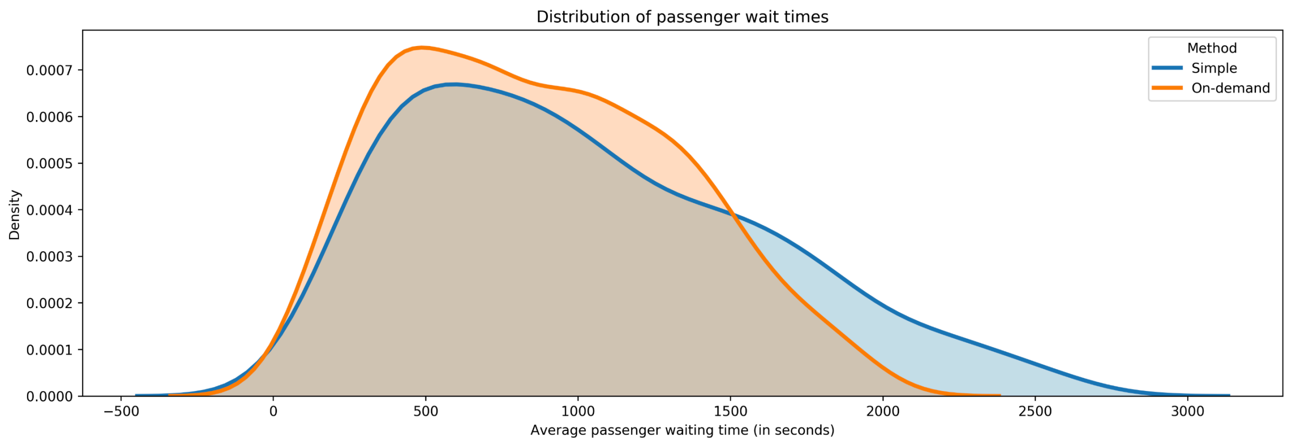




This graph indicates the transitions to floors of the elevator over time as it runs the scenario. Each vertex indicates when doors are opened and closed, with the darkness of the color indicate how much of the elevator capacity has been exhausted. We can see that in this scenario, the “on-demand” elevator strategy has finished the task much earlier (around 442 seconds) than the “simple” strategy (around 520 seconds). In the Jupyter notebook, the user can hover over each vertex (like on the bottom figure) to view how many passengers are the in elevator and which passengers are entering/exiting the elevator.

Single scenario is only an anectodal evidence for efficiency improvements. Next, we will look into how the two metrics we use are distributed across randomly generated scenarios. To do so, for each run, we generate a new bottom and top floor for the building and randomly generate a set of random number of passengers. Each generated set is run on both elevator strategies. The elevator capacity is fixed at 8 passengers maximum. Here are the results:





The top figure shows the distribution of elevator run times on random sets of building size and passengers and the bottom figure shows the distribution of average waiting time of passengers in different passenger sets. In both metric distributions, we can observe that the orange distribution ("on-demand" strategy) has a smaller mean of 1758 seconds than the blue distribution ("simple strategy") whose mean was 2155 seconds. In the bottom graph we also got better results for the "on-demand" strategy: 868 seconds vs 1023 seconds ("on-demand" vs "simple").

Interpretation

We saw that the "on-demand" elevator has an edge over the "simple" elevator both in terms of the total time spent by the elevator as well as the waiting time for an average passenger. This means that overall, the "on-demand" elevator is more preferable, whether we want to decrease the waiting time of an average passenger or if we want to decrease the waiting time of the longest-waiting passenger (indicated by the elevator run time). To understand how much better the "on-demand" elevator does, we can plot the distribution of ("on-demand" run time minus "simple" run time) and see if it is always negative ("on-demand" elevator always faster) and how big is that improvement on average.

Contributions and learnings

In this assignment, Khoi's main contributions were:

- Assist in brainstorming the functionality of the three classes
- Communicate the requirements and outputs of methods
- Building's simulation functionality and logging
- Building's passenger and elevator state management
- Passenger's randomized elevator entry time
- Interactive granular level plots
- Code debugging

Khoi has learned about dividing states and state management across classes in an intuitive manner. He received a good revision on how randomization in simulating and comparing the behavior of two different algorithm. He also learned to do interactive plots using `bokeh` for the first time.