2015

Elevator Report

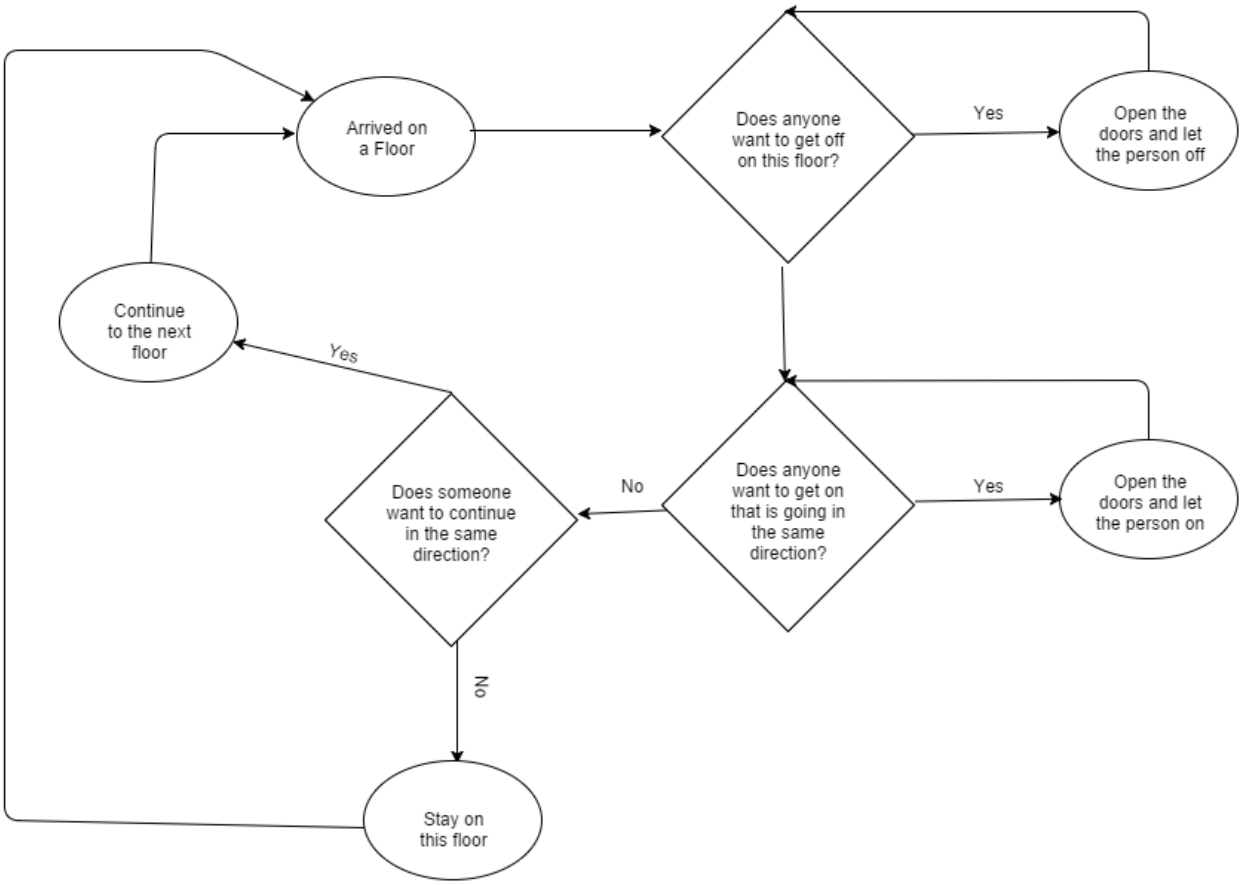
RYAN HARTMAN, EZRA PERRY, LUCAS DOS SANTOS

Motivation

Our motivation for this project was to create an algorithm to optimize the operation of an elevator in a building. This algorithm can be used for a way to improve elevator performance for real life usage. As a group, we decided that it would be best to minimize the amount of time each person waited between their initial button press, and the time it took to reach their destination floor. For us to be able to program an elevator to function like this, we first researched into the current way elevators work and are designed.

Research

From our research we found that the typical elevator design was one where while an elevator currently has passengers, it will continue moving in the same direction. On each floor it will stop and open the doors if anyone wanted to get off, pick up anyone currently on that floor that wanted to move in the same direction, and continue on until the elevator was empty, or until no one wanted to continue in the same direction. At this point, the elevator would then begin moving in the opposite direction and repeating this same process. The main point where elevators differ is what to do when no requests currently are available and the elevator is empty; however, most have the elevator continue to idle on the same floor till the next request is found. The general elevator algorithm can be represented by this flow chart:



This algorithm would then form the basis for our advanced elevator that we would improve upon. For the base elevator, we created a much simpler algorithm to test against. The basic elevator would continue moving in the same direction, stopping on each floor and opening the doors allowing anyone to get on the elevator regardless of the direction they needed to go in, and allowing anyone to get off once their destination was reached.

Elevator implementation

To begin, we decided to first create an implementation of our basic elevator, and the standard algorithm that is normally used for elevators that we found from our research. We then ran simulations to see just how much quicker the normal elevator was when compared to the rudimentary elevator. We created multiple situations to test the elevators in and measured the time taken between a person's

request, and when they were dropped off on the floor the needed. These test cases, and way of measuring performance will then be used to compare the advanced elevator implementation with both the basic and "rudimentary" elevator

Rudimentary Elevator Implementation

For the rudimentary elevator, a queue and an ArrayList were the only data structured required for the implementation of the elevator. A queue was used to store all of the requests that the elevator will receive during the simulation. The elevator will only be able to see and respond to requests that occurred after the current time in the simulation so that it is not able to use future information in its decisions. The ArrayList was used to store the current passengers that were on an elevator. An ArrayList was chosen over an array because the elevator does not have a set maximum number of people able to fit on it. Instead it has a maximum weight capacity so a fixed size data structure would not have worked. For the algorithm itself, first the elevator opens its doors. Then it sees who is on the elevator and if anyone wants to get off on this floor. Once it has let off all the passengers that have reached their destination floor, it then checks through the requests in the serving queue that it knows of. If any of the requests are from the current floor the person is added to the elevator. The elevator then shuts its door and proceeds to move to the next floor. Once it reaches the top floor or the ground floor, it changes directions and the process repeats itself.

Basic Elevator Implementation

The basic elevator required a similar implementation to the rudimentary elevator but the logic is more advanced. For data structures, a queue and two ArrayLists were needed. The queue and one of the ArrayLists served the same purpose as they did in the rudimentary elevator implementation. The second ArrayList was added to store the requests the elevator is currently aware of. This was added to help with the logic of which floor the elevator moves to next. The algorithm for the elevator first begins

by finding which requests the elevator is currently aware of and moving them from the queue of requests into the ArrayList used to store known requests. The elevator then checks the current passengers and sees if anyone wants to get off at the current floor. If they do they are removed from the ArrayList storing elevator passengers and the time they were released is recorded. The elevator then checks through the requests it's aware of to see if anyone on the current floor wants to get on the elevator and if they are travelling in the same direction that the elevator is going in. This is one of the main improvements over the rudimentary elevator in that passengers only board when they are travelling in the same direction. This helps the elevator to continue to have free space for requests as it continues to move throughout the building. After picking up all valid passengers from the current floor, the elevator then decides which floor to move to next. The elevator first checks to see if it is empty, and if it currently doesn't know of any requests. If this is the case, it idles on the floor it is currently on. If the elevator has passengers on it, it then continues to move in the same direction that it was currently travelling in. If the elevator currently has no passengers but knows of requests, it moves to the oldest request it has received. This helps to minimize the wait time for each person as the oldest requests are serviced first once the elevator is empty.

Elevator Comparison

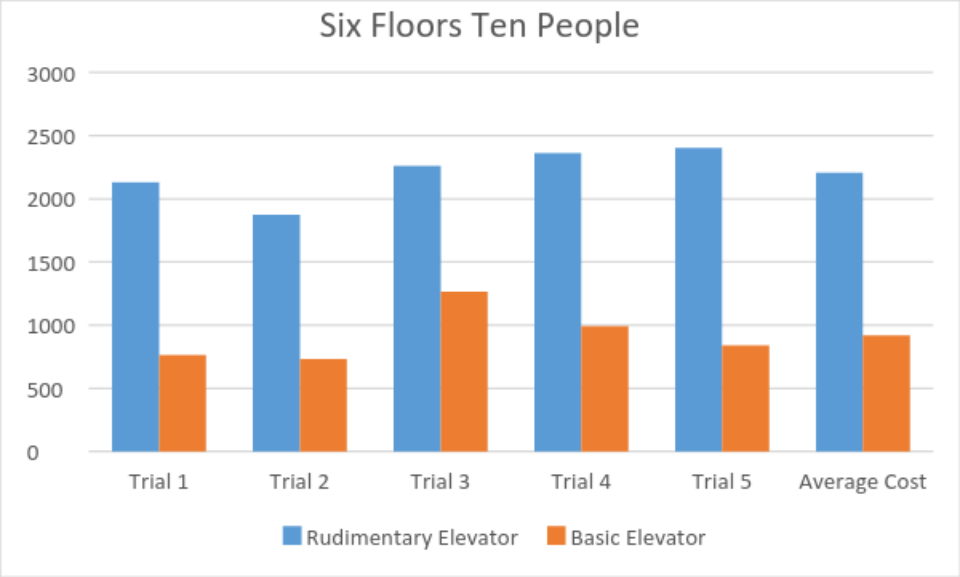
To compare these two implementations, we developed a series of test cases to simulate how the elevator would perform in a real life situation. The test cases we developed were as follows:

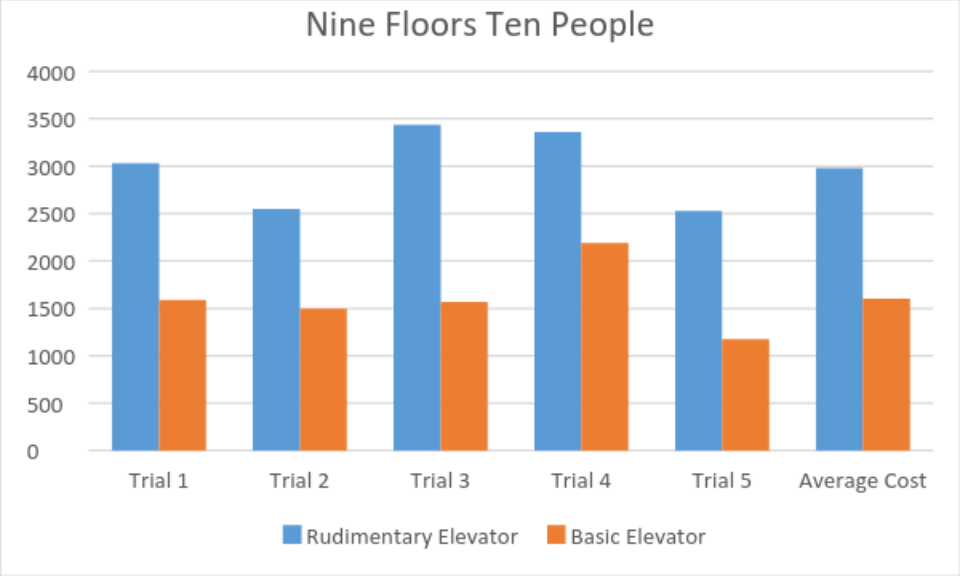
- Six floors with 10 requests
- Nine floors with 10 requests
- Thirty floors with 10 requests
- Ten floors with 10 requests

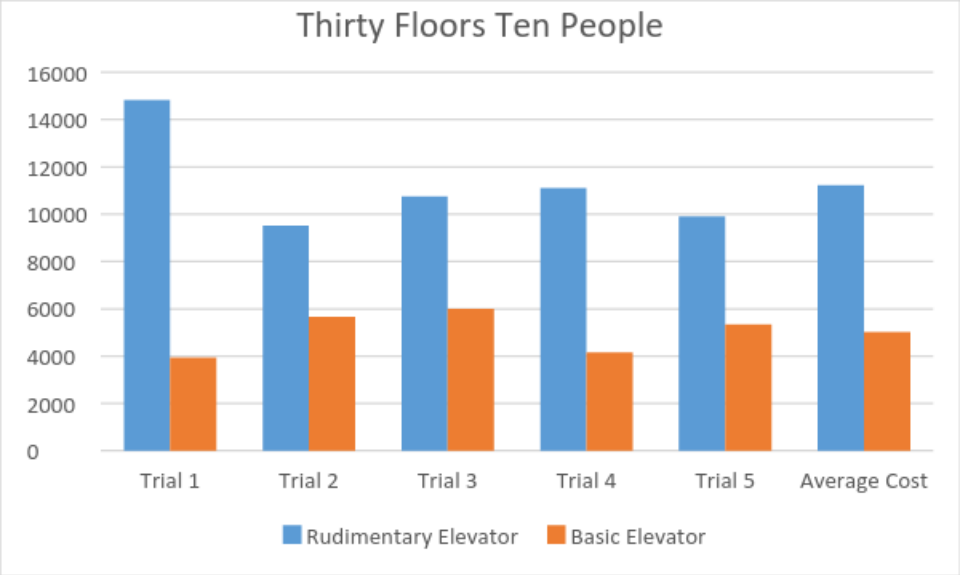
- Ten floors with 50 requests
- Ten floors with 100 requests

All of the test cases were doing using a seeded random number generator to make sure that both elevators had the same data to perform against. All tests cases were also simulated five times to make sure that the chance of outliers in the total cost of the elevator performance were reduced.

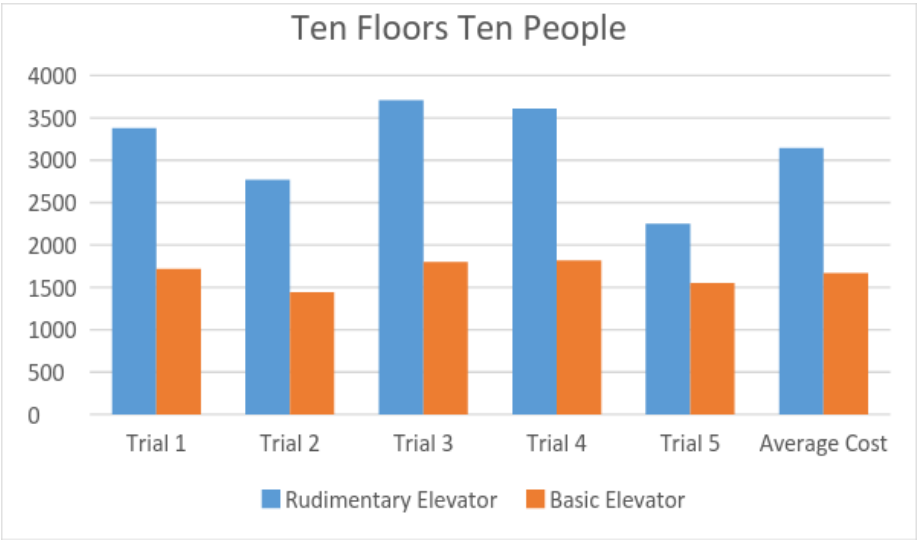
Results

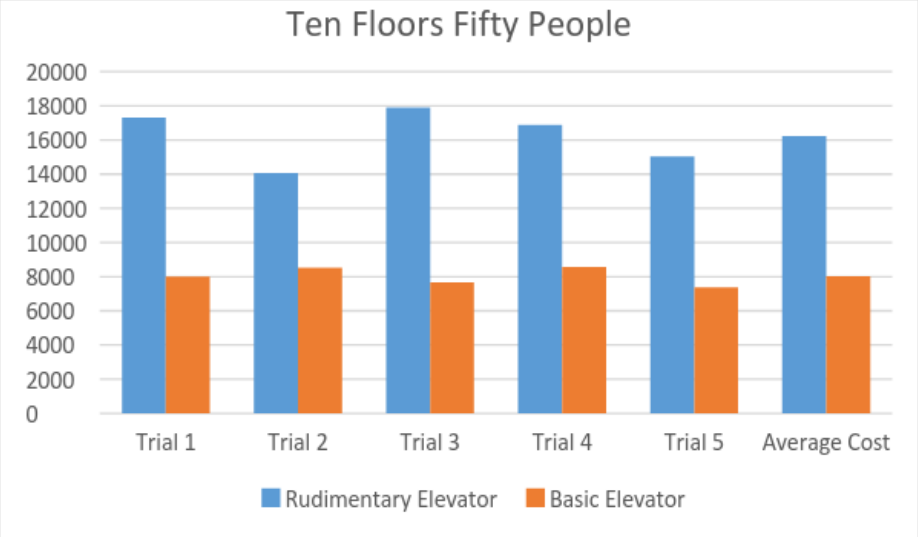


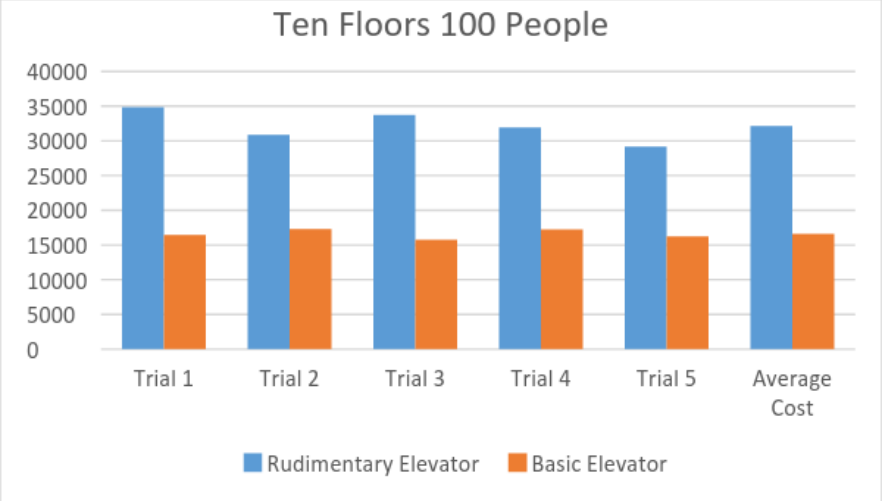




Based off of the results from keeping a constant number of elevator requests while only varying the number of floors, the basic elevator algorithm performs approximately twice as fast under all test cases. We then tested keeping a constant number of floors and varying the amount of passengers to see if this relationship still held.







When the amount of floors is kept the same and the number of people varied the trend still continues where the basic elevator design is approximately half of the cost of the rudimentary elevator design. Now that we have a comparison for the rudimentary elevator compared to the base elevator design, we were able to see the performance of it and begin to look for ways to improve the algorithm for operating the elevator.

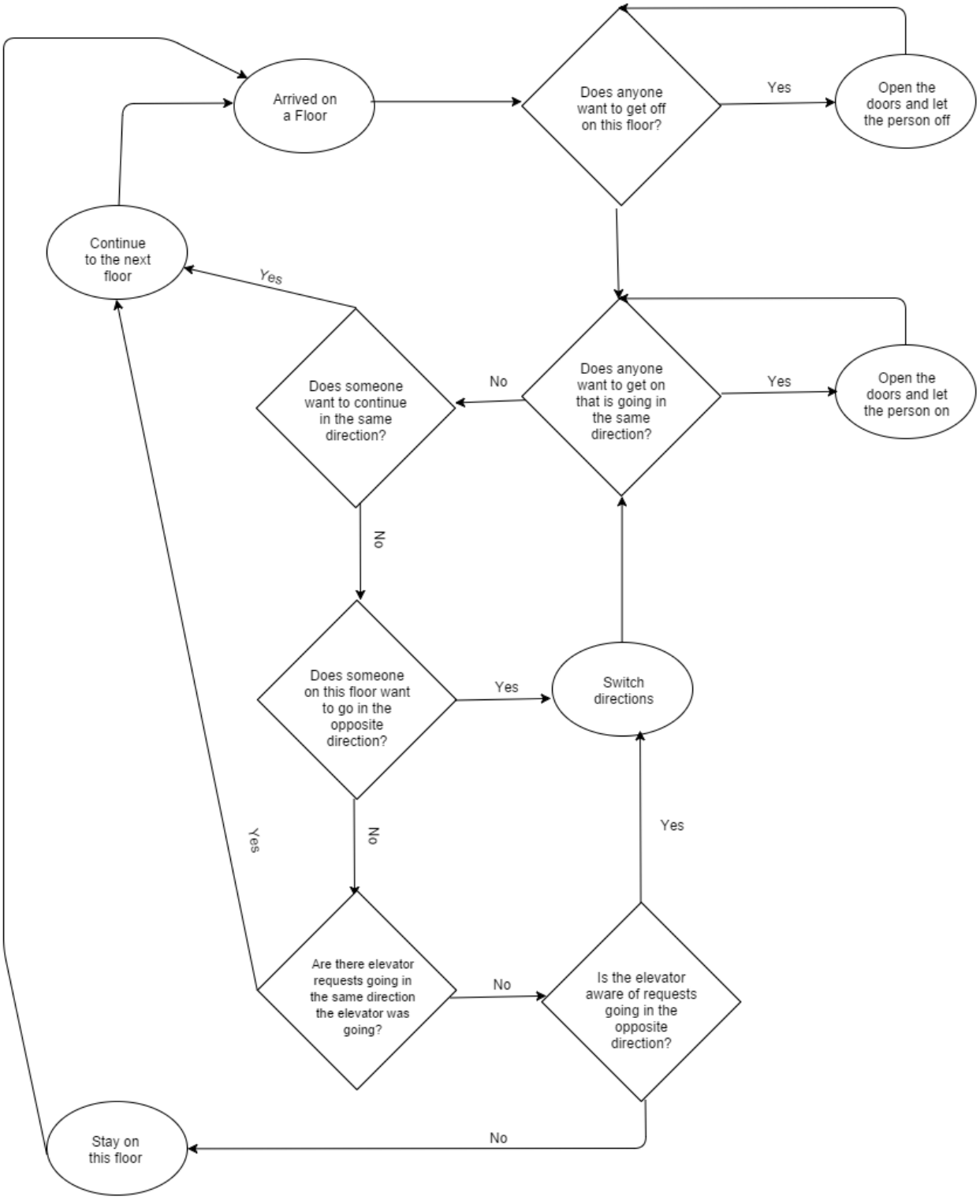
Improving the Elevator

After analyzing the results between the rudimentary elevator and the basic elevator implementation, there was no clear area in which the basic elevator appeared to struggle. It consistently performed around twice as fast as the rudimentary elevator design and there was no clear improvement that could be made. However, when watching our visualization, we noticed one key problem with the elevator. If the elevator was empty and moving in a direction to the oldest request, if the request needed to go in the opposite direction that the elevator was travelling, the elevator would not pick the person up and instead continue moving in the same direction. This was causing a noticeable slowdown in servicing all of the currently known requests. We were able to solve this problem by checking if the elevator was empty and if a request wanted to go in the opposite direction when attempting to pick up a passenger. If this was the case, the passenger was then picked up and the elevators direction would then be reversed. The other improvement that was made was in what the elevator would do when there were no passengers but the elevator was aware of requests. We changed it so that the elevator would move to the oldest known request so that this would lower the overall wait time for all passengers.

Advanced Elevator Implementation

The implementation for the advanced elevator required a queue, an ArrayList, and a PriorityQueue. The queue served the same purpose as it had in the basic and rudimentary elevator implementations and the ArrayList continued to be used to store the current elevator passengers. The main change for this implementation was changing storing the known requests from an ArrayList to a PriorityQueue. This was done so that the first element would always would always be the oldest known request so that it could be serviced as soon as possible. The algorithm itself for the elevator remains mostly the same with only two key changes as discussed above. The algorithm begins by checking to see

if any of the passenger requests have occurred that the elevator is not aware of. If one occurred before the current time, it is then offered to the PriorityQueue. The elevator then compares all of the passengers currently on the elevator and checks to see if any of them want to get off on this floor. If any passengers do, and the elevator door is not opened, the doors are opened and the passengers that want to get off on this floor are let off. The elevator then checks all of the requests it knows of and if the elevator currently has passengers on it, it picks up any passengers moving in the same direction. If the elevator is empty, and there is a request to move in the opposite direction, the elevator picks up that person and switches direction. The elevator then continues to move in its current direction so long as it has passengers. If the elevator is empty it moves towards the oldest known request. If there are no known requests, and the elevator is empty it idles on the current floor. The algorithm can be represented by the following flow chart:

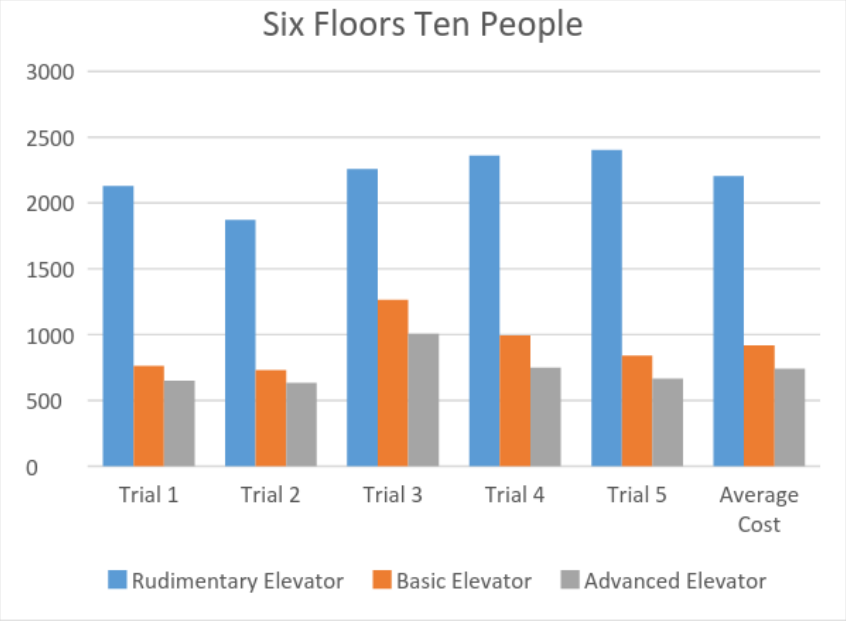

Arrived on a Floor
Does anyone want to get off on this floor?
Yes
Open the doors and let the person off
Continue to the next floor
Yes
Does someone want to continue in the same direction?
No
Does anyone want to get on that is going in the same direction?
Yes
Open the doors and let the person on
No
Does someone on this floor want to go in the opposite direction?
Yes
Switch directions
Yes
No
Are there elevator requests going in the same direction the elevator was going?
No
Is the elevator aware of requests going in the opposite direction?
Yes
Stay on this floor
No

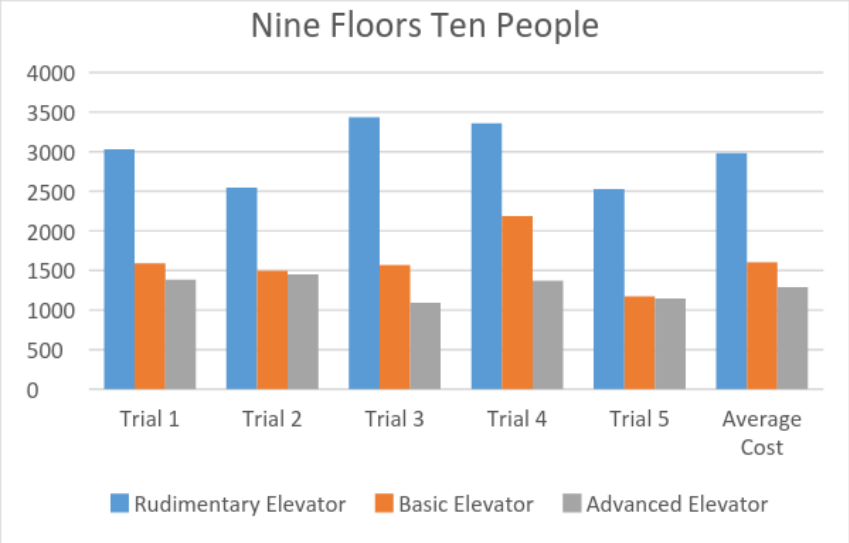
Performance of the Advanced Elevator

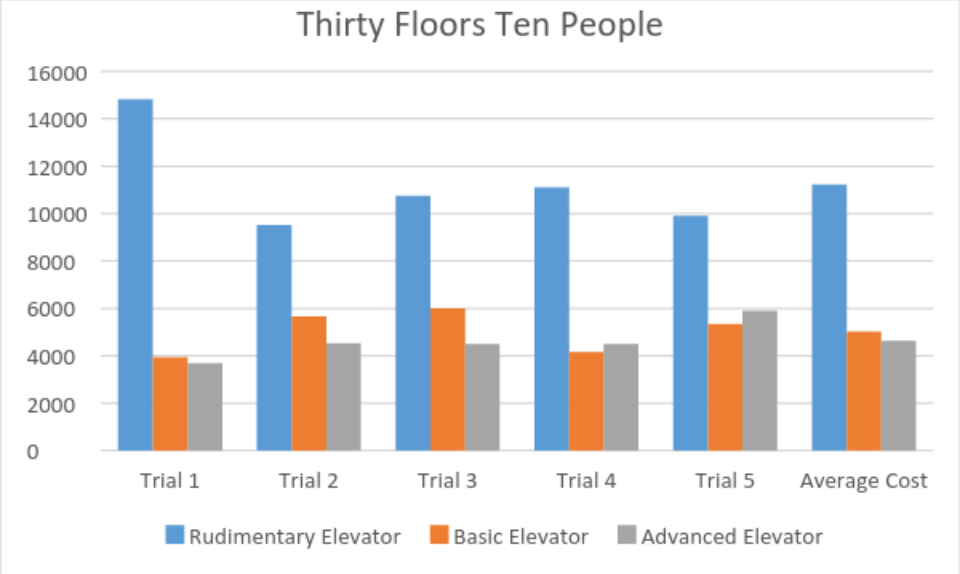
Now that we have our advanced elevator implementation we can compare its performance to that of the rudimentary elevator algorithm, and the basic elevator. The elevators will be compared on the same test cases to see if there is still a performance increase.

Results

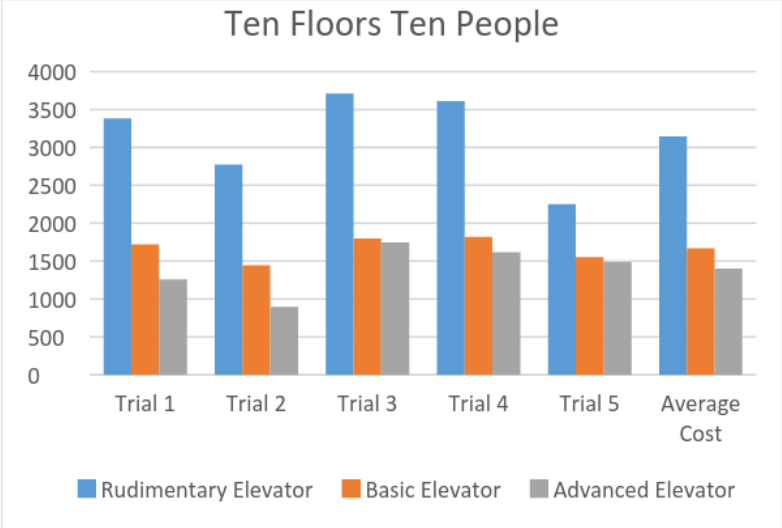
Keeping the amount of passenger requests constant and varying the amount of floors, the costs were as follows:

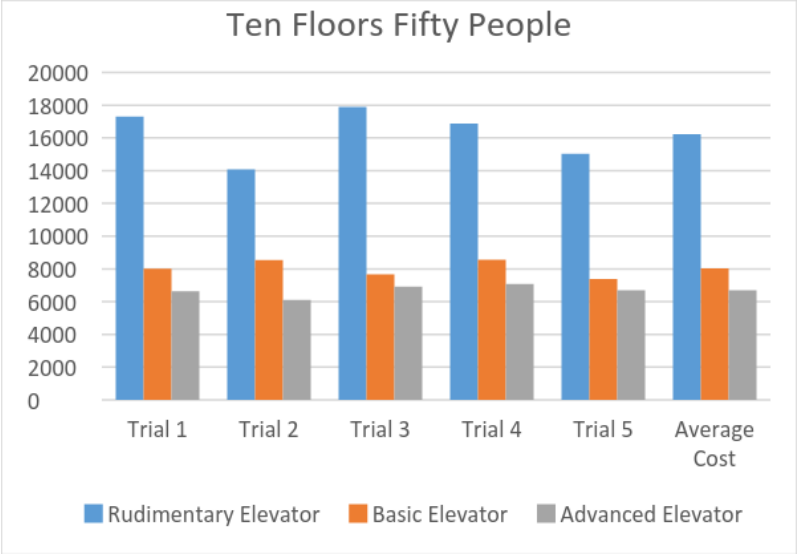


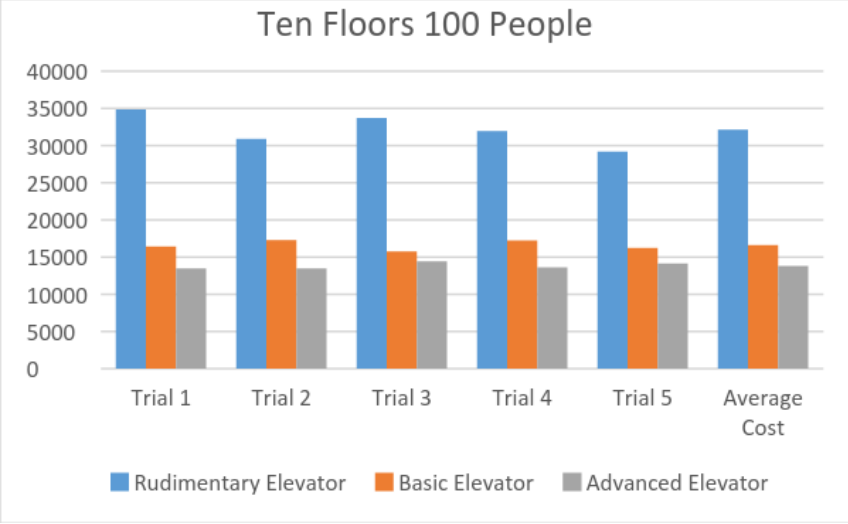




For the majority of the test cases, the advanced elevator showed a noticeable improvement over both the rudimentary elevator. The basic elevator implementations performed, on average, 295 seconds faster than the basic elevator in the total cost for servicing all of the requests. We then tested cases with a constant number of floors, varying the amount of passenger request to see if the advanced elevator design continued to outperform the basic and rudimentary elevator implementations.


Ten Floors Ten People
4000
3500
3000
2500
2000
1500
1000
500
0
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average Cost
Rudimentary Elevator
Basic Elevator
Advanced Elevator


Ten Floors Fifty People
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average Cost
Rudimentary Elevator
Basic Elevator
Advanced Elevator


Ten Floors 100 People
40000
35000
30000
25000
20000
15000
10000
5000
0
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average Cost
Rudimentary Elevator
Basic Elevator
Advanced Elevator

The advanced elevator implementation, again, outperforms both the rudimentary elevator and the basic elevator. Here there is a much larger improvement once that amount of passenger requests is increased with the advanced elevator implementation performing 1345.4 seconds faster on average than the basic elevator with ten floors and 50 people, and performing 2779 seconds faster on average than the basic elevator. This shows that while the advanced elevator performance isn't much faster depending on the amount of floors, when the amount of passenger requests increases it has a much lower average cost for servicing all of the requests.

Improvements for Real Life Implementation

The advanced algorithm implementation we came up with provides a good start for how to design an optimal elevator for use in real life; however, there are still improvements that could allow for a better implementation and further reduce the overall wait time for all passengers.

One improvement that could be made would be to change how the elevator behaves when it is empty with no current requests. The implementation could instead have the elevator idle at different positions depending on the time of day. For instance, in the mornings the elevator could idle on the ground floor since most requests will be people heading up to wherever they need to go, or idling on one of the middle floors during the day so that it is equally close to all floors and able to service all floors in a similar amount of time. These would help reduce the wait time of the elevator arriving to pick passengers up.

Another improvement for real life situations would be having more than one elevator to service requests. The implementation for the elevators themselves would stay the same, what would change is which requests are assigned to each elevator. One implementation for this would be to have whichever elevator is moving in the same direction pick the elevator up if it is going to pass it, and if it is not going to pass it, have whichever elevator is closest and moving in the correct direction pick the passenger up.

Weekly Logs

Change Logs and Project Progress

8/18/2015: Met with group and discussed who would be working on what aspect of the project

Ryan: Code PDF, and API

Lucas: Presentation

Ezra: GUI

8/25/2015: Meeting to check/evaluate progress. Ryan began constructing a basic concept of the project which allowed Ezra to begin conceptualizing a GUI with the assistance of Lucas.

9/15/2015: Completion of basic elevator. GUI still in progress. Change logs created.

9/20/2015: Code reworked to account for misinterpretation of door timing concept.

9/24/2015: Group meeting to evaluate progress. Project still being worked on.

9/29/2015: Completion and implementation of GUI by Ezra and Lucas.

10/1/2015: Completion of improved Elevator.java.

10/11/2015: Generic improvements to GUI of Elevator project.

11/10/2015: Lucas and Ezra began work on the PowerPoint for the presentation. Ryan began work on the PDF with help from Ezra.

11/15/15: Completion of PDF. GUI finalized by Lucas. Code finalized by Ryan.

11/20/2015: PowerPoint completed. Code slightly reworked to improve timing.

References

https://en.wikipedia.org/wiki/Elevator

https://en.wikipedia.org/wiki/Elevator_algorithm

http://www.i-programmer.info/programmer-puzzles/203-sharpen-your-coding-skills/4561-sharpen-your-coding-skills-elevator-puzzle.html

https://courses.cit.cornell.edu/ee476/FinalProjects/s2007/aoc6_dah64/aoc6_dah64/