Elevator Project
Brent Allard and Zachary Paryzek

The Objective

Our goal was to minimize the movement of our elevator, while still providing reasonable service times for passengers.

The GUI/Architecture

While there are many files and classes, the fundamental structure of the project is as follows:

- Brain: 'Brain' itself is the parent of the three classes, DumbBrain, SmartBrain, and PatientSmartBrain, that can be used to control the elevator's actions. This exists separately from the actual elevator.
- ElevatorController: This class is responsible for nearly all GUI functions, and contains a reference to the Brain and ElevatorInfo (more on that later) that are instantiated in Main. Additionally, the control flow of the program centers around this class, which also handles low-level things like unloading PassengerRequests from the main Queue to the many Queues in the Building when their time comes, and incrementing/decrementing the ElevatorInfo's currentFloor value.
- ApplicationStartGUI: This class exists so that whatever Main is used for testing doesn't have to extend 'Application', and do whatever else is required for JavaFX to work. It is used when the ElevatorInfo.operate() method is called.
- ElevatorInfo: Inherits from the provided Elevator class. This contains all of the information the elevator has access to, and also holds the queue/queues (more on that later) of PassengerRequests.
    - Data Structures: Because any passenger within the elevator may need to be be removed at any time, and any button within the elevator may need to be depressed/unpressed at any given point in time, and accessing these two pieces of information always involved creating an iterator and passing through every value, LinkedLists were used.
- Building: This is a class instantiated and contained within ElevatorInfo, representing everything that is happening outside of the elevator car, like buttons pressed, and where people are waiting. Methods like hasUpRequest(int floor) make code in the Brain and ElevatorController simpler.
    - Data Structures: Because the size, in floors, of the building is known before Building has to be instantiated and it is not always necessarily the case that we need to iterate through every value, arrays and ArrayLists rather than LinkedLists to hold the following:
        1. An ElevatorRequest enum for each floor

2. A Queue of PassengerRequests at a given floor that are waiting to go up, and a Queue of PassengerRequests that are waiting to go down

- Params : Contains static, final values which are drawn from to generate random samples, and to hold the values for the constructor in our own Main class. This allows us (and you) to change just about anything, including which brain our elevator uses, quickly and easily.
- PasengerRequest: We added a generateSample() method which takes in the sqlTime that the elevator begins operating, and then uses other data in the Params class to generate a continuous, random sample of requests which was used for testing.
- Print: Once the elevator finishes operating, this class is used to write information about each of the PassengerReleased objects to a file, like the example .txt file on Canvas
- Shell Scripts: for ease of use, we created clean.sh, compile.sh, and run.sh, which we encourage you to use when running our project. Simply replace 'Main.java' in run.sh with whatever main you use for testing
- TimeManip: sqlTime is alright because you can compare times, and there's a useful toString() method build in, but it fails in other ways. TimeManip is used to generate Time objects that are a certain number of seconds after a given Time, to calculate the difference, in seconds, between two times, and to convert a Time to its absolute value in seconds (and vice versa).
- Cmd: this is an enum used to represent commands which the brain sends to the ElevatorController to be executed.
    - wait – causes the elevator to wait for a very short time, specified in Params
    - up – causes the elevator to move up one floor
    - down – causes the elevator to move down one floor
    - open – opens the doors
    - close – closes the doors
    - start – the command is passed when the 'start' button in the GUI is pressed
    - finish – this command is given when the continueOperate() function becomes false

Control flow:

1. Application starts, Elevator is instantiated, etc…
2. GUI is launched.
3. User presses 'Start' button (passes Cmd.start)
4. ElevatorController's 'update' method is called. ElevatorInfo values are updated, and Queues are managed according to the current time. If the doors are open, Passengers enter and leave the elevator.
5. (Still in update method) Control is delegated to the brain, which decides what to do next based on ElevatorInfo's information. The brain passes this Cmd back to ElevatorController, which plays the appropriate animation

6. Once the animation completes, ElevatorController's update() function is called again.
7. Repeat 4-6 until update is called and continueOperate() evaluates to false, at which point a loop that has been running in the ElevatorInfo object is broken, and the ArrayList of PassengerReleased is returned.

- Note: This architecture uses multithreading, and calculations are performed in real time. This is a more accurate representation of what an actual elevator control system might look like, but suffers from the inability to shut the GUI off. Unlike a system in which all calculations are performed first, then a GUI is launched to display what happened, the time required to calculate what to do will affect the time values of the elevator. For example, if it takes .5 seconds to move from one floor to the next, but it also takes .1 seconds to make that decision, .6 seconds will pass before update() is called again.
- This architecture makes a few assumptions:
    - Every floor has an up and a down button, but the up button on the top floor and the down button on the bottom floor will never be used.
    - If the elevator door opens, and there are people are waiting to go both up and down outside of the elevator, the only ones who will enter are the ones who want to go in the direction of the button that was just depressed
    - People will leave the elevator the moment the door opens on their desired destination

Brain Algorithms

Dumb Brain: This is the base elevator that moves all the way from the bottom of the building to the top, then back down again, picking up and dropping off people as it passes along.

Pseudocode:

- Open doors if a button inside of the car is pressed for the current floor, or there is an external request to go in the same direction that we are currently going (up button pressed, and we happen to be going up)
- Close the doors if they're open
- if (at top floor), then we're going down
- else if (at bottom floor), then we're going up
- if (going up) go up
- else if (going down) go down

Performance:

To check whether to open doors should open, we need to iterate through every internal button pressed. Because this calculation is performed before all others, and because all other calculations are simply conditional statements, this algorithm runs THETA(i), where 'i' designates internal buttons pressed

Smart Brain: Improvements are made to dumb brain to eliminate wasteful movement, but the elevator still makes long passes along the building, changing directions only rarely.

Pseudocode:

- if the doors are open, close them
- If there are no requests in or outside the elevator, just wait
- If (at top/bottom floor OR no buttons pressed in the direction that we're currently going), change direction
- If (there is an internal request for the current floor OR (there's an external request in the direction that we're going AND we can probably fit their weight)), open the doors
- If (going up) go up
- Else if (going down) go down

Performance:

Checking whether to open the doors involves iterating through all internal buttons pressed, while checking whether to change directions involves iterating through some fraction of all the building's floors, and checking if there are no requests involves iterating through EVERY floor. However, if the doors are open, the computation only takes linear time. For this reason, the algorithm runs in O(i+f), where 'f' is the number of floors, and 'i' is the number of internal buttons pressed. Because i <= f no matter what, this simplifies to O(f)

PatientSmartBrain: This brain attempts to optimize for economy of movement by waiting a certain amount of time, or until a certain percentile of the buttons are pressed before moving, then waiting again once a certain number of passes (characterized by movement, then eventually a change in direction) are completed.

Pseudocode:

- If (waiting)
    - If (percentage of buttons pressed > Params.serveCondition)
        - Stop waiting
    - Else if (Params.timeToWait seconds have passed since we began waiting)
        - Stop waiting
- If (not waiting)
    - If (no requests)
        - Start waiting
    - Else if (we've changed directions >= Params.maxDirectionalChanges times)
        - Start waiting
    - (if we're still not waiting) follow logic of SmartBrain

Efficiency

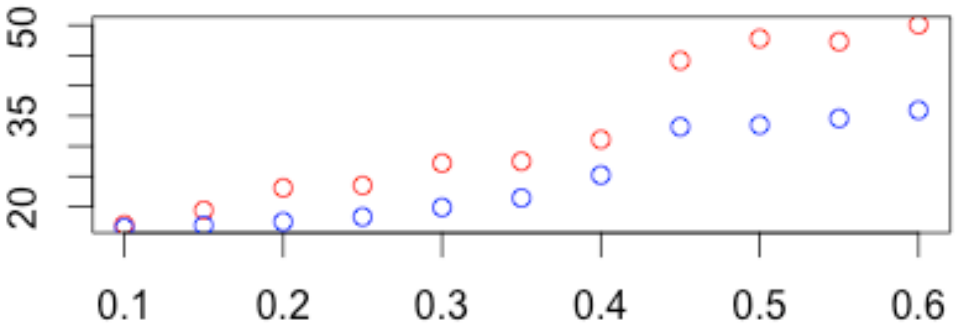
Even if we're waiting, to see if we should stop waiting requires checking every single floor of the building for requests, and we already know that SmartElevator runs in O(f). The one time we perform a linear calculation is when we BEGIN waiting, so we can still say that PatientSmartBrain is O(f), not THETA(f)

Data analysis:

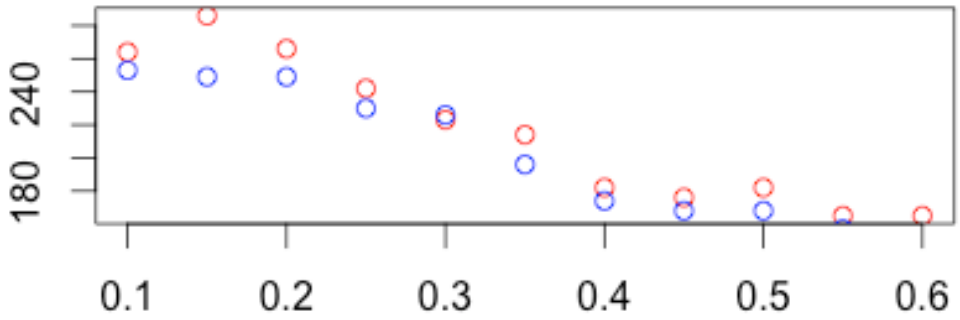
Optimizing Serve Condition:

Constant values:
Floors = 10 Test Duration: 180 People In Test: 60 Max Waiting Time: 60
Max Directional Changes: 3

Serve Condition (x) vs Mean Wait Time (y):
Red -> Queue from seed = 2
Blue -> Queue from seed = 1



Serve Condition (x) vs Total Moves (y):
Red -> Queue from seed = 2
Blue -> Queue from seed = 1



It can be observed that, in terms of minimizing movement, x = 0.4 is the point at which we reach diminishing returns. Interestingly, it is also the point at which waiting times increase far more rapidly!

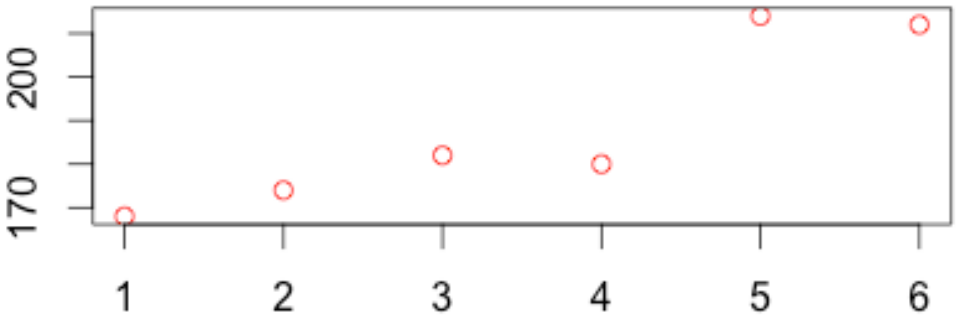
I attribute this to the nature of the serving condition, that it cannot actually tell how many people are waiting, but must rely on the buttons being pressed. With a uniform continuous distribution like the one we're using, as the overall fraction of buttons pressed increases, the odds that a new person waiting for the elevator will press a button that has not yet been pressed decreases.

Conclusion: 0.4 is the ideal value for the Serve Condition

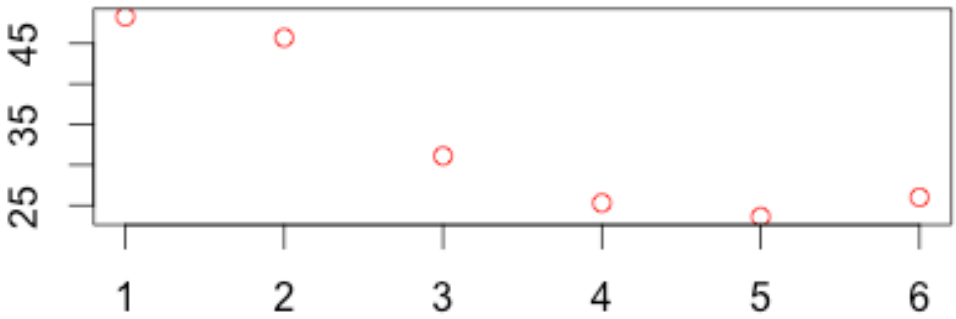
Optimizing Max Directional Changes:

Constant Values:
Floors = 20 Test Duration = 180 People in Test = 60 Max Wait Time = 60
Serving Condition = 0.4 Seed = 2

Max Directional Changes (x) vs Total Moves (y)



Max Directional Changes (x) vs Mean Wait Time (y)



From x=1 to x=4, the mean wait time is more than cut in half, while the total moves required to deliver only increases marginally (~7%). Beyond that point, however, the number of moves performed increases significantly, with little to no return in terms of wait time.

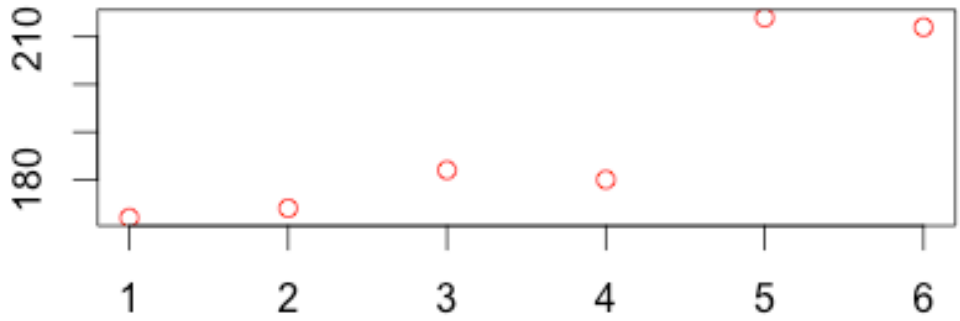
Conclusion: 4 is the ideal value for Max Directional Changes

Optimizing Wait Time Duration:

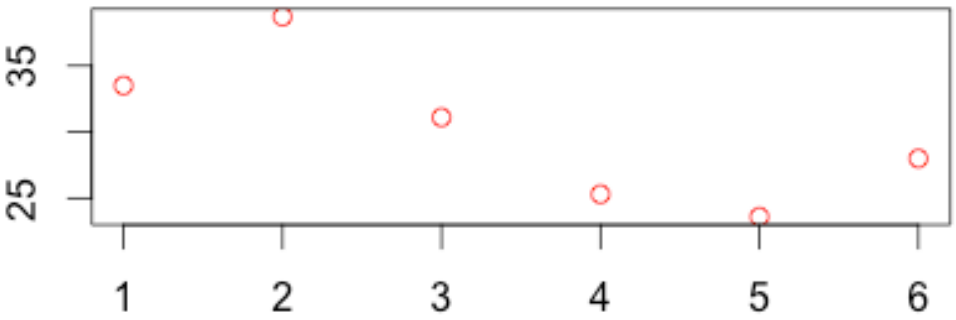
Based on the previous page, we've optimized the number of 'runs' across the elevator per wait time of 60 seconds, resulting in a ratio of 1(runs):15(seconds) ratio. However, is it better too keep that ratio while using shorter wait intervals and fewer runs per interval?

Constant Values:
Floors = 20 TestDuration = 180 People In Test = 60 Seed = 2
Ratio (described above) = 1:15

Max Changes In Direction *_**or 1/15 wait time!**_ (x) vs Total Moves (y)



Max Changes In Direction *_**or 1/15 wait time!**_ (x) vs Mean Wait Time (y)



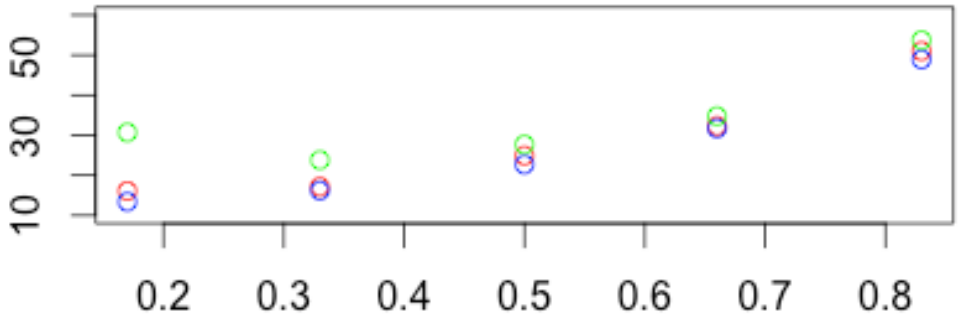
Conclusion: Altering Max Changes in Direction and Wait time while keeping the ratio the same does have an impact on wait time and total moves, but the values we were already using (4:60) appear to be optimal.

The Final Test: How does our brain stack up?

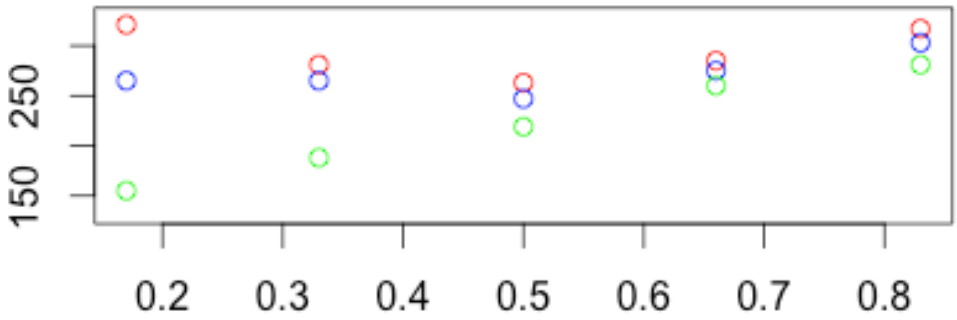
Here, we measure mean wait times and total movement for each of the three brains at a variety of 'congestion' values (on average, how many people are making requests per second). Because we're using a constant time interval for these tests, it's the number of people served which must change.

Constant values:
Floors = 20 Test Duration = 180 Time To Wait = 60 Serve Condition = .4
Seed = 1001

Red-> DumbBrain
Blue-> SmartBrain
Green-> PatientSmartBrain
Congestion (x) vs Mean Wait Time (y)



Congestion (x) vs Total Moves (y)



Conclusion:

The PatientSmartBrain has very good movement efficiency at lower congestion values, but as congestion increases, the PatientSmartBrain's behavior approaches the SmartBrain's behavior. The SmartBrain, in turn, is slightly more efficient *and serves passengers faster* than the DumbBrain, but this advantage also degrades as congestion increases.