Project 1: Optimizing Elevator Movement

Zoe Hackshaw

Summary

In summation, this code used 10 random integers between 1-100 (including the top and ground floors) and to represent 10 random people on 10 different floors. Each person was assigned two floors, the first one being where they started, and the second one being the floor to which they are going. After being assigned, the code used MCMC methods to choose two people at random, propose a change in floors, and then evaluate the distance between these floors. The code then proposes a change, or a swap in floors, and evaluates if the change results in saving distance (good) or creating more distance (bad). In the positive section, only positive swaps were accepted causing the answer to regularly converge. In the annealing section, sometimes bad swaps were accepted for the optimization of the entire code, which resulted in irregular spikes but also caused the answer to converge, sometimes more accurately than the positive section.

Code Architecture

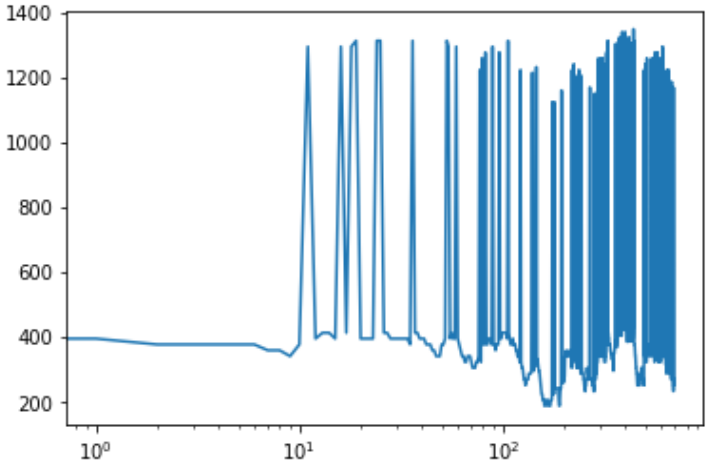
Firstly, I import all of the modules necessary for this project, with the notable ones being `random` and `randit` from `numpy`. `Randit` enables us to choose random integers, which will come in handy when choosing floor numbers, because they have to be integers. I then defined the number of floors and people using `nfloors` and `npeople` respectively. Two functions were defined to make the calculating of distances easier. The `mag` function was created to take the magnitude of the two positions in an element, which is one person going from their starting floor to their ending floor. THis was done by taking the square root after squaring the x and y values. The `distance` function was defined to take the magnitude of the distance from the first person to the second person. I created an empty 2 dimensional array defined as `r`, then used that array to assign two random floors each to the 10 different people. To do this, I utilized a `forloop` and the `randit` function to assign random floors between 1-100. The first element is the floor where the people are on, and the second element is the floor that people are going to. In the code, I printed r to ensure that the array works as needed. After this, I created an empty array using `dsaved` to create an empty array, and calculated `D` using the distance function for the initial distances without optimizing anything.

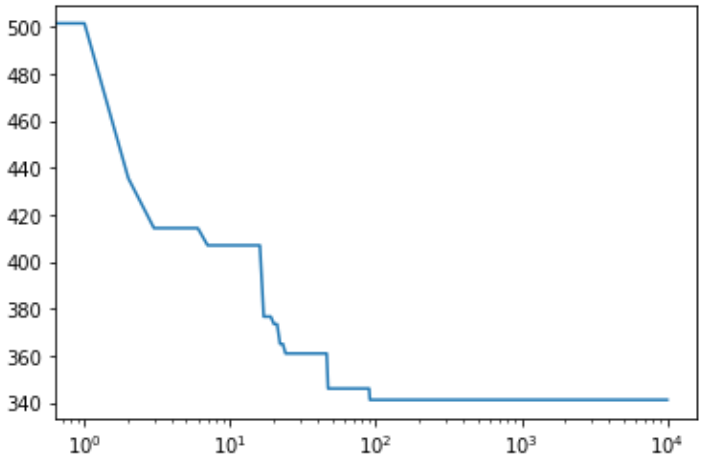
The next stage of the code is creating a method where only positive swaps were accepted. I looped through the entirety of the following code first 1000 times, but after I completed that the answer did not converge, so I increased it to 10,000 and the code worked. In this forloop, the first thing that I did was pick two random people, notated by `i` and `j`. There is a while loop for the case that i and j might be the same value, this is then fixed by just assigning them new values. I then renamed our old D as oldD because that will be redefined later. Then, to swap the cities, the command `r[i,0],r[j,0] = r[j,0],r[i,0]` is used, along with the same code for the second argument. `D` is then recalculated with the same distance formula, and `deltaD` is determined as the distance between `oldD` and the new `D`. If `deltaD` is negative, that means it takes less distance, and is a good swap. Thus, a positive `deltaD` is a bad swap, so the if statement determines if `deltaD` is positive, and swaps `r[i]` and `r[j]` back while

appending the recalculated `D` to `dsaved`. The else statement just automatically appends any good swads to the array.

The last stage of the code incorporates annealing, which is sometimes accepting bad swaps. This essentially follows the same overall structure as the positive swaps, with minute changes. Five new variables are introduced, which are needed for the annealing process. Tmax is the maximum number in degrees which we then want to cool to Tmin, the minimum number. T is the time, t is a time constant, and tau is a constant. To cool the code, first we increment time with `t= t+1`. We then set `T=Tmax*np.exp(-t/tau)` to have our code cool exponentially. These numbers were originally derived in the book, and then they were played with for optimization. During the cooling is where we then go through the entirety of the MCMC method that was outlined in the positive swaps section.

Discussion

- Originally, before I got my code to work, I ended up with the plot pictured below. While this was not correct, it helped me realize that I originally had the forloop m range through 1000, but I needed to increase it by an order of magnitude to 10,000.
  - 

- For the code that accepts only the positive swaps, I plotted the graph below. This code decays regularly and eventually converges to 340.
  - 

- For the annealing code, this graph has a lot peaks and troughs, yet the answer converges at 230, which is more efficient than the code that accepted only positive swaps. Annealing works!
  -