ECM1414 write up

153539

September 7, 2020

1 Algorithm explanation

The base case is that of a lift that can make no decision but merely goes up and down changing direction at the top and bottom. My proposed algorithm works off a priority-based system where the algorithm assigns each floor a priority to be served and then serves the highest priority floor. This has a number of benefits, of particular note is that this system is easily expanded just by adding a few lines of code. For example, if you wanted the CEO's floor to always have priority when they wanted to use the elevator, just multiply the lift priority by 100 for their floor and they will always be served first. This new unique rule could be added in less that a minute with a single line of code on installation making this a highly customisable algorithm.

Whist any of these rules for priority can be added or taken away at the leisure of the customer, the lift comes with a few base rules for the normal and orderly running of the lift:

1. Each person has a priority that increases by 1 every time step – Whilst in the simulations everyone will have equal priority this is important in real life as it ensures that no one will ever be waiting forever as their priority will eventually definitely be higher than all others.

2. The time steps required to do something act as a divisor on the priority – This means that in order for it to be worth going twice as far, under normal conditions, twice the people are waiting to be served.

3. Impossible actions contribute no priority – So the lift won't give any priority to the people who want to get on if it is full.

4. Getting off is twice as important as getting on – This helps prevent the lift from filling up as it will focus on serving the people in it first.

5. Once the lift decides on a floor to serve it gains twice the priority – This is designed to prevent the lift from going up 1 floor then back down then back up in a loop. Whist this will not be observed in the simulations, if people are arriving in real time it is important that is doesn't randomly turn around when it is nearly at its destination and this would induce inefficiencies.

6. If actions have equal priority the highest floor will be served – This was selected as the resolution as there are always a limited number of people on the upper floors, as the only way into the building is the lower floors (in normal buildings) and, thus it will eventually serve people at the top enough that the lower floors take priority again.

7. The lift will know only the number of people on each floor the way they want to go and the floors the people in the lift want to go to – This is because for most lifts only after people step into the lift, the button is pushed.

8. If something goes wrong and all the actions are considered impossible then the lift will revert to basic routing and print a warning – This prevents the lift from stalling if the algorithm ever fails. It should be noted that I have not observed it to ever fail in testing as yet (except when inducing it to ensure the reversion to basic routeing worked).

My lift planning algorithm and simulator also make certain assumptions that assist user experience. For example, the algorithm assumes that the lift will open on every floor as it passes allowing anyone en route to the target location to get on/off meaning no one will ever see the lift fly past their floor. I also assume that people will queue outside the lift and as a result will get on whether or not the lift is going in their direction instead following a queue structure. This means that my algorithm is less optimal than if people only got on if it was going their direction, but in practice many people would ignore it anyway so in a real setting there is no significant loss.

Finally there exist a few configuration constants at the top of the source code to make minor adjustments to the importance of different factors. For example the constant DISTANCE_NEGATIVE_IMPORTANCE which defaults to 1 effects how much the elevator should care about the distance of a floor from it. This allows further customer optimisation and customisation.

2 Performance analysis

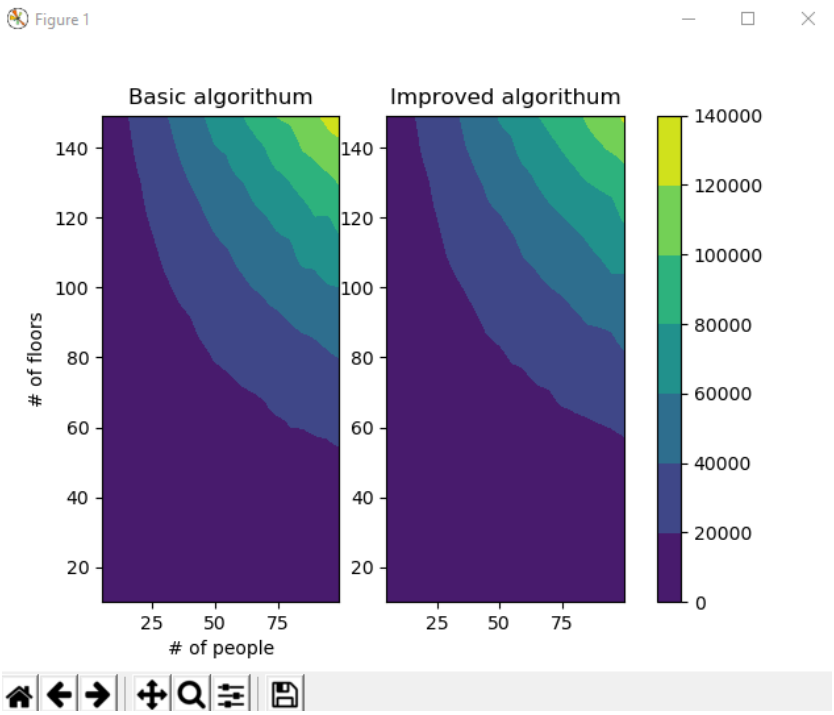
In order to demonstrate the effectiveness of my algorithm when compared with the base case I have created a system to simulate elevators. The core of my system is the simulate functions which when called will begin a simulation of the passed elevator. It has many options/arguments allowing the simulation to be finely controlled, notable among them is the option 'mode' which allows one to specify either 'step' or 'solve'. In 'step' mode the simulation can be shown on a graphical interface allowing a clearer understanding of what is going on. In 'solve' mode the simulation runs as fast as it can allowing the result to be found quickly, this is good for testing en masse.

2.1 mass testing

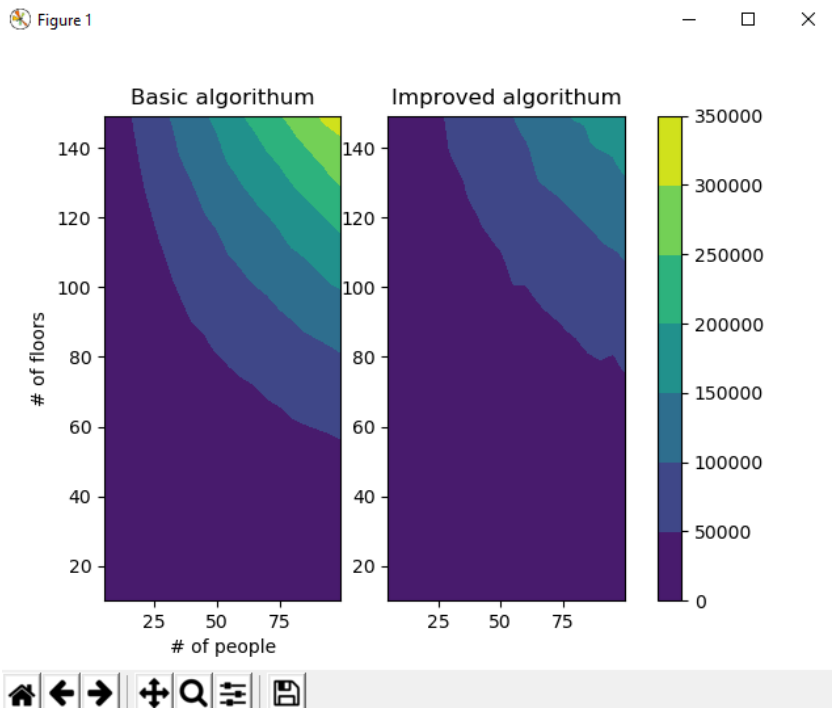
To mass test the function batch has been supplied which allows the batch testing of algorithms with variant inputs over ranges. It returns the results of the test as a dictionary which, via other functions (seen in example 2b at the bottom of the code), can be plotted to see how algorithms perform in a graphical contour plot.

2.2 results

The first plot performed it with entirely random distributions of people going to entirely random floors:



Whist there is not a huge benefit (about 10% better) this is to be expected as people don't actually want to go to random floors and thus the algorithm was not designed with this data in mind. Quite often one floor will be far more popular; for example, the canteen may be on floor 5 and thus around lunch time half the people in the building want to go to floor 5. This sort of situation is where my algorithm excels. Below is a simulation of the end of the day where about 80% of people want to go from random floors to the ground floor (other people may be stopping off elsewhere to finish things or cleaners coming in after hours).



This result is clearly exceptional, it performs on average twice as well and this is even better than it looks at first; as the start and end of the day are the times that the elevator is busiest. That is the exact time that the elevator being very efficient is important. Its minor benefits over the base case the rest of the day are fairly unimportant compared to this as elevators are designed for peak usage rather than for the minor random traffic throughout the day.[1]

3 Data structures

The core data structure to my implementation is an associative array containing all the information about the state of the lift and building. This was selected over tuples, lists and similar sequence data types for 2 reasons. Firstly, the order isn't important and would be picked arbitrarily. Secondly using an associative array makes it much easier to interpret the data just by printing it as rather than having to remember what the number in index 4 means you can see the word "lift floor" by it.

My algorithm assumes that people arrive at an elevator call terminal in an order and that they will get on in the same order they arrived. For this I used a queue, implemented with a list, as python has no alternative besides a tuple. The list was chosen over the tuple as people need to be added and

removed from the queue and tuples being immutable would be a poor fit for this task.

4 Video example

Here is my video demo. Whilst I managed to find some software that would capture my screen with multiple applications at once and record my voice at the same time it made my voice's pitch randomly change and my editing software couldn't do much to help so I uploaded it as it is.

`https://www.youtube.com/watch?v=c2zM826Ys98&ab_channel=jb1159UOE`

5 Weekly log

- Week 1: created graphical displayer for the lift, created cost function, created basic planning algorithm
- Week 2: created solve mode for the simulator and created the control buttons for the GUI
- Week 3: made the generate function, made advance planner
- Week 3: first test of algorithm – always better than the base case so far but not much $\approx 10\%$
- Week 3: - may need to rethink this plan my algorithm isn't great it is even as bad as the base case if there are large elevators
- Week 4: - Ok done some thinking and some testing, my algorithm is fine for any normal elevator set up (as most elevators have a smaller max capacity than the areas where my algorithm performs poorly) and I have just tested my algorithm with a skewed distribution of people and if 80% of the people want to go from a random floor to the ground (like at the end of the day) then my algorithm is nearly twice as good as the basic planner. This means I will highlight this fact in my sales pitch pdf.
- Week 5: finished batch function
- Week 5: learnt how to use matplotlib – contour plots and scipy interpolation to display 3d coordinates and wrote a function to use it. Coding functionality is now complete more or less.

- Week 5: Went back over the code adding doc-strings to functions
- Week 6: class/module doc stings + comments
- Week 7: Finally found some software that could record what I needed and made the video
- Week 8: added my weekly log and did a final look over everything before submitting

References

[1] Apex-Elevator. What to consider before arriving at number of elevator/lift required and capacity of elevator/lift. Retrieved from `http://apexelevators.com/what-to-consider-before-arriving-at-number-of-elevator.html`, 01 2017.