Lift Coursework

Student Number: 690009073

1 Introduction

The mechanical implementation of the lift system is very simple and inefficient. My lift system will aim to reduce the average wait time for a person in the lift. I will simulate the lift system running over a range of a different number of floors, and people. There will also be a graphical representation of the system, but for the analysis I will use the version without the graphics as this will allow the program to run faster.
My new algorithm will improve the current system in two ways:

- Reduce the number of floors the lift has to cover.
- Split up the lifts to maximise how many people they carry at once.

I will implement the system in python, using the libraries:

- tkinter
- matplotlib

The tkinter library will allow me to give a graphical representation of the system, and the matplotlib library will give information on how well the system works, providing a way to compare the base system and my own implementation. These libraries are both well documented, so it will make them easy for me to use. [1][2][3]

2 Lift Control System Details

In this scenario, a given number of people are distributed randomly over a given number of floors, and a given number of lifts will serve the people and deliver them to the correct floor. For this reason, there needs to be scope for the user to enter the number of floors and people. Therefore, before the system starts, the user will be asked to input the number of each they would like to simulate. In the simulation which will generate my graphs, a range of values will be used for the number of floors and people. I have decided to include 6 separate lift cars in the system. The capacity of the lift will be 10. [4]

2.1 Basic System Details

In the basic lift system, the specified number of lifts start at the ground floor, then move up to the top floor, then to the bottom in a loop until all the people are delivered. If a person wants to travel upwards, then they can only get on the lift if the lift is travelling upwards. Because of this, the lifts will always travel together, and they never split up.
Effectively, adding more lifts to the basic solution doubles the capacity of the lift, and does nothing else to increase how fast the algorithm can run. Therefore, in a scenario where the average number of people per floor is small, there will be no need for a lot of lifts, as the first few lifts are never at capacity. It would be much more efficient if the lifts could split up and serve different floors at once. Also, when the lifts travel from top to bottom in a loop, often some floors do not need to be visited. By finding a way to cut out these floors that do not need to be visited, the algorithm will run much faster.

2.2 Improved System Details

In my improved system, I will implement a more efficient algorithm. I aim to minimise the average wait time for people in the building.
As detailed in the previous subsection, there are two main problems with the basic system that my new algorithm will tackle:

1. Lifts should find a way to split up
2. Lifts should not go to floors that do not need to be visited.

All the lifts need to start at the ground floor and move upwards, but they still need to split up. To solve this, I decided to assign each lift an initial maximum floor, which they will go to before turning round and resuming the algorithm as normal. To determine this initial floor, I used the formula:

$$\frac{nf}{l} \tag{1}$$

where n is the lift number (the order in which the lift objects are instantiated), f is the number of floors, and l is the number of lifts.
This will make sure that the lifts are split up. The problem that this creates is people who start on the low floor who want to end up at the top getting in the lift too early. For example, if a lift is initially capped at floor 5, but it picks up someone on the ground floor who wants to go to floor 10, then this person has to stay in the lift for 20 floors, whereas in the basic implementation they should only be in the lift for 10. This is not only a 10 floor increase in this person's wait time, but also it could stop 10 people entering the lift if it is running at capacity. Therefore, I will make sure that people can only get in a capped lift if their destination is lower than the initial cap.

To make sure that floors that do not need to be visited are not visited. I will set a maximum and minimum floor for each lift, which is updated regularly. To find the maximum floor for example, set the maximum floor to 0 (ground floor), then iterate through the floors, and every time someone has a start or destination greater than the current maximum, set the maximum to this value. Then, loop through the lift's contents and do the same. This is explained further in section 3.2. When the lift reaches this floor, it will turn around. Potentially, this could cause each lift to have a different maximum and minimum as their contents are different. This is a good thing, as it causes the lifts to split up further, and especially towards the end of the simulation, it will help the lifts to pick up the last few people faster. This method of only picking up people if they want to travel in the same direction as the lift is used in modern lift control systems. [5]

The complete new lift algorithm will work as follows:

1. For each lift in the building:
    (a) Find an initial maximum floor.
2. While there are still people to be served:
    (a) For each lift in the building:
        i. If the lift has not reached its initial maximum, the maximum = initial maximum, and minimum = 0
        ii. Else find the maximum and minimum floor for the lift
        iii. Drop off the people in the lift whose destination floor matches the current floor and are going in the same direction as the lift.
        iv. Pick up people on the floor until the lift capacity is full.
        v. If moving up or down is impossible (it is bigger than the maximum or lower than the minimum) turn around.
        vi. Move up or down.

Of course, this requires the lift system to 'know' what floor the passengers want to get off on. In the traditional circumstance, the passenger will press 'up' or 'down' to call the lift. In my implementation, the passenger will have to press a button with their intended destination on a central console on the floor, and the system would tell them which lift to get on.

3 Data Structures and Algorithms

3.1 Data Structures

Lift Contents:

The lift itself should be an array, because it is a direct access structure with a fixed size. However, in python, arrays are implemented as lists, so I had to add a limit to the size of the list which I will have to use for the lift. [6]

Lists of People on Each Floor:

I decided to use a dictionary to store each list of people. The keys would be the floor number where the people are waiting, and each person object is stored in a list once again, as it needs to be direct access, but this time of variable length.

Storage of Simulation Data

For this, I used a dictionary consisting of lists of integers. After each run of the simulation, I added the average wait time, the number of floors and the number of people to a text file through a function in the lift class. When I unpacked this data at the end of the simulation, they needed to be stored so that they could be plotted in the graph. I felt that the best way to do this was to create a dictionary with fields 'people', 'floors' and 'waittime' which would store each of the lists.

The Building

In my program, the building is an object. One of its attributes is 'floor_queues'. This stores the people in the building, divided up into floors. To do this, I used another dictionary, this time a dictionary of lists of people objects. The key of the dictionary is the floor, so to access the first person on the 5th floor, use building.floor_queues[5][0].

3.2 Algorithms

Searching:

In order to find the maximum and minimum values in the building, the whole dictionary of people had to be searched. Because there is no order to the lists in the dictionary, a linear search had to be used which is quite inefficient (as opposed to a binary search). In reality this would not be an issue, as even though it is slow, it is not as slow as a lift travelling up a building, and could easily be calculated as the lift was travelling, or while people were being loaded.

Loading People:

To find the people in the list that need to get on the lift, the following algorithm is used:

1. For each person in the list:
    - (a) If the lift is not full:
    - (b) Look at the next person in the list and find their destination.

        i. If it is larger than the current floor, and the lift is travelling upwards, load the person.
        ii. If it is smaller than the current floor, and the lift is travelling downwards, load the person.

To drop off the correct people in the lift:

1. For each person in the lift:
    (a) If the person's destination is the current floor, remove them from the lift

To generate the building:

1. For n=0 to the number of floors required:
    (a) Create a new empty list in the building class's floor_queues dictionary, with a key of n.
2. For i=0 to the number of people required:
    (a) Generate a random number x between 0 and

Finding the maximum and minimum floors for a lift:

1. Set max = 0 and min = the number of floors.
2. For each floor in the building:
    (a) For each person in the floor's dictionary entry:
        i. If the person's destination is greater than max, set max = their destination.
        ii. If the person's destination is less than min, set min = their destination.
3. For each person in the lift's contents array:
    (a) If the person's destination is greater than max, set max = their destination.
    (b) If the person's destination is less than min, set min = their destination.

4 Performance Analysis

To analyse the performance of each algorithm, I ran each one over a range of 10-100 people, and 10-100 floors with a step of 10. This means that I have 910 data entries for each algorithm. I plotted each one on a graph, and also had my simulation print the mean value of the averages, giving a very easy comparison between the two systems. The averages were found by taking all of the people in the building, adding up their wait times, and dividing by the total number of people. Then the overall mean was found by adding up all of the wait times and dividing by the number of people.

4.1 Basic Implementation

In order to analyse the performance of the system, I first needed to view the performance of the basic system. Here, we can see that there is not a great deviation between average wait times with a small number of floors. However, when we reach the 50 floor and above range, the points become far more scattered. We can also see that decreasing the number of people causes little change. If anything, it makes the algorithm worse. The points become more scattered, and the maximum average wait time is produced with a small number of people. The matplotlib interface shows that the maximum wait time is around 135. The system prints out that the overall average wait time is 62.4.

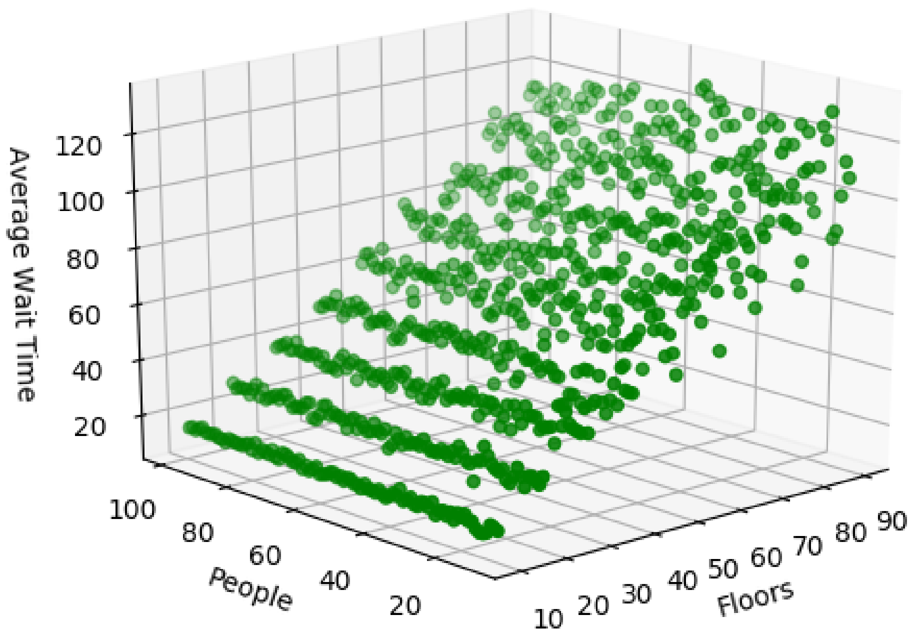

Figure 1: Basic Simulation

4.2 Improved Implementation

In the graph for the improved simulation, we can see the same trend of less deviation in the lower floor numbers, and more spread out data in the higher end. This is because there are many ways in which the building can be generated, so it is likely that a better and worse cases are generated when there are so many simulations run with a small change in number of people. In this graph we can also see a different shape than the basic graph. The average wait time increases from the region with a small number of people and a large number of floors to the region of large number of floors and people. This shows how the new algorithm does not waste the extra lift capacity like the basic algorithm does, due to the splitting up of the lifts. Overall, the wait times are lower, even in the region with a large number of floors and people, where the basic algorithm can make use of the extra lifts more efficiently. This is due to the maximum and minimum limits being set so that the lift does not have to waste time visiting unnecessary floors. We can see that the maximum average wait time for the improved algorithm is around 105. This is an improvement of 22.2%. The overall average wait time for the improved system is 49.8. This is an improvement of 20%.

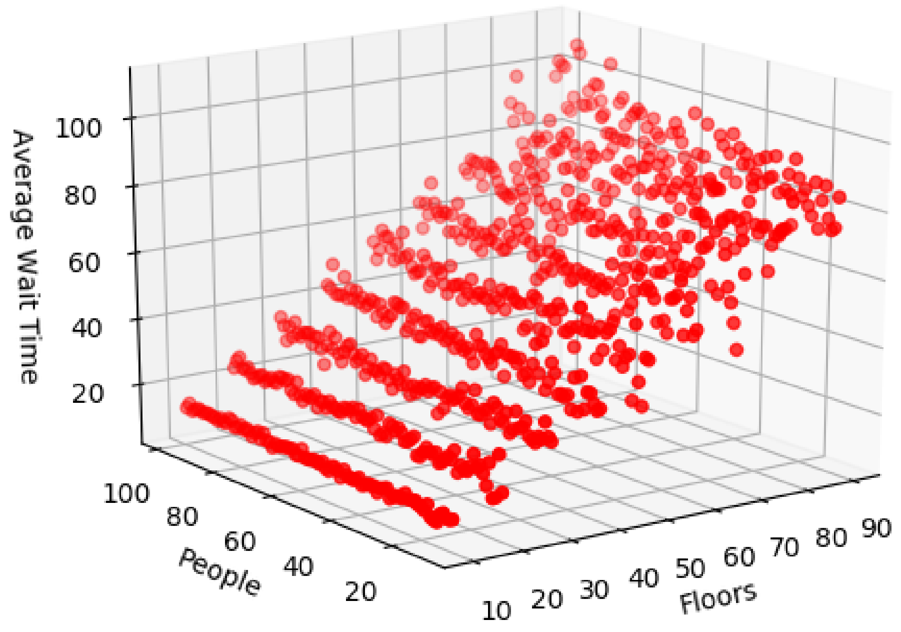

Figure 2: Basic Simulation

5 Video Recording

A link to a video recording of my system:

6 Weekly Progress Log

20/01

Started to plan my solution. I thought about the pros and cons of implementing the solution in Python and Java, eventually deciding on Python. I am more familiar with python, and I decided in a project where the GUI is quite important but the algorithm is where my time should be more focused, that I should use the language in which the GUI would take the least amount of time. I then considered the libraries I would use and how to use these to create the lift. I chose tkinter and matplotlib as I am familiar with both from previous projects.

27/01

Began to implement the basic solution. I started by making an outline of each class I would use in the program, deciding on a class for a person, lift and building object. I think that an object oriented approach to the problem is better suited, as I can make good use of polymorphism to deal with person and lift objects, such as a general method 'move' which is called on each lift to move the lifts up or down a floor.

03/02

Finished the basic solution, making people 'get on' the lift by adding them to an array stored as an attribute of the lift class. I had a few problems with this, mostly due to the way the building was updating the graphics, so I had to refactor the code quite significantly which took quite a while. The basic solution is now implemented, with a full graphical representation.

10/02

This week I created a new version of the basic solution so that I can keep track of the wait time of each person. This involved adding some attributes to the person class and a counter to the building class which is updated whenever someone with a greater maximum wait time leaves the lift.

17/02

Throughout this week I tested some new ideas for the new lift. I was looking for something simple to implement, but something which has a strong effect on the wait time. I tested ideas such as restricting the lift to deal with sections of the building, trying to clear out lower floors first so that it can serve higher floors quicker and trying to narrow down the number of floors it has to serve on each trip up or down the building.

24/02

Having settled on an idea from last week, I began implementing the new algorithm. This was quite a lot easier than I thought it would be, which is probably due to the the basic system being in place already. This means I only had to refine the basic solution, changing small sections of code which would change the operation of the lift. This included adding in some methods to the lift class such as 'find_limits' which return the maximum and minimum floor for the lift so that it doesn't have to visit floors which do not need to be visited (as there is both nobody waiting on the floor and nobody needing to go to the floor).

02/03

Now that the new algorithm was implemented, I needed to start adding ways to compare the two systems. In the first week, I decided on using matplotlib to graph the operation of the lifts as it was a library I had some experience in. During this week, I added a text file that the system would write to every time a simulation was run, so that I could start building up some data on each system. I also created a new version of each system. In this version, there are no graphics, and the simulation is run over a variety of floors and people combinations. This produces a detailed graph which I can use to compare each system in different areas, for example not many floors but lots of people, or a large number of floors and people.

09/03

Made some improvements to the GUI. Before it looked quite messy. The new GUI has the same information on it, but it looks much cleaner. This was quite new to me, as I had never really experimented a lot with making tkinter windows look better than the default look.

16/03

The deadline was extended this week. The university also closed for COVID-19, so I spent this week packing to move out, so did not get very much work done.

23/03

After some clarification via email to another students question, I realised the algorithm I had implemented is not appropriate for the specification of this coursework. This means that I need to think of a new algorithm and implement it, rewrite the whole write up, and make any necessary adjustments to the GUI. The new algorithm is the one that will be detailed in this report. I spent this week trying out new algorithms, based in multiple lift and single lift systems until I found one that should work. Before settling on this multiple lift system, I had ideas including 'daisy-chaining' the system (picking up people based on the floor previous passengers got off on), eliminating the top and bottom floors first, then the 1st and second to top floors and so on until the whole lift had been cleared, and changing the direction of the lift based on how many people are above and below it. However, none of these would have been both simple enough to implement and fast enough to beat the original system.

30/03

I redesigned the basic implementation this week to make use of multiple lifts, and remade the simulation mode for the basic system. This took more time than expected, because for some reason the wait time counting was not working properly, and when I included multiple lifts in the system, it had would decrease the speed of the system, though clearly it would make it faster when I watched the graphical version. I had to change the whole counting system, because before the wait time was updated in the lift's move() method, but now that there were multiple lifts, every time the move() method was called, the wait time would be increased by every lift, so if there were 6 lifts, for example, the wait time would have been recorded as 6 times what it should have been.

06/04

This week I implemented the new algorithm, making use of the new basic system to build upon. This turned out to be quite simple, with relatively few issues. I then made the simulation mode for the new algorithm, and began to make improvements to make it faster. One of these included not picking up passengers who needed to travel higher than the initial limit for each lift, which increased the performance a lot.

13/04

I finished up the write up, and made some small adjustments to the GUI to make it look better. I then made the video of the system.

7 References

[1] - Tkinter Documentation - https://docs.python.org/3/library/tk.html
[2] - Effbot (Tkinter tutorials) - https://effbot.org/tkinterbook/
[3] - Matplotlib Documentation - https://matplotlib.org/3.2.1/contents.html
[4] - https://www.building.co.uk/cpd/cpd-17-2017-specifying-lifts-in-commercial-buildings/5089794.article
[5] - https://www.popularmechanics.com/technology/infrastructure/a20986/the-hidden-science-of-elevators/
[6] - https://docs.python.org/3/tutorial/datastructures.html