08/01/20

Evaluated the guidance concerning our Data Structure and Algorithms Coursework and started planning some of the various features and functionality that would eventually make it into the final project.

Different Elevators Having Different Speeds

Different Elevators having overlap between certain floors

Different elevators having varying capacities

Service elevators being used for fire evacuation of a building

Start of the day having several elevators at the ground floor

Priority of different floors

Example of a hotel with bottom floor reception

Living quarters in-between

Penultimate floor has a gym and sauna

Top floor has a swimming pool

Questions you should be asking yourselves (not limited to these)

• What do you want to minimise?

• Wait-time of people? (recommended)

• Do you have another idea? Ask me or the TAs

• What is the max capacity of the lift and how do you control for this?

• How is the lift called?

• Two-button system: up and down (recommended)

• What if the building has multiple lifts?

• Start with 1 lift!

Baseline is a lift that can only change direction when it reaches either the top or bottom floor of the building. Start with the case where you have a building with only 10 floors.

Be sure not to keep people in a permanent status of waiting.

Graph for evaluation of efficiency using number of floors of movement to serve one user and the number of floors this user is being moved.

Random generators function to simulate people travelling within the building.

Cumulative weight of all users in the elevator, stops the lift from moving if overcapacity.

15/01/20

Started working on a simple user interface using PyQt which would allow the user to input the number of lifts as well as the range of floors that each individual lift has access to. This would later link to a tkinter animation showing the lift moving up and down between floors. This would later be extended to include random passengers moving between floors to test the efficiency of the system.

23/01/20

Designed a simple animation with a square moving up and down representing the lift movement.

Later I will add floor numbers and allow the lift to stop at each floor to allow passenger on and off.

The first version of my lift animation generates a lift shaft which stretches from the top of the canvas to the bottom. Inside there is a lift which moves smoothly up and down and changes direction whenever it reaches either end of the lift shaft. Currently much of the positioning and sizing elements are generated using numerical data this will later need to be changed such that the animation can adapt dynamically to the size of the canvas, number of lifts and number of floors.

25/01/20

After having experimented for a while with the various animation styles I came across a method to split the canvas into multiple cell forming a grid structure. This would allow me to change the colour of each cell individually representing the movement of a lift between floors. I started with generic code used to generate a checkers board of square cells which change colour if clicked by the user. The script was modified to allow it to be resized and updated with a different number of column and rows which would later represent the number of floors and number of lift shaft within the building. From here I was able to create alternating columns the first of which contains the floor number and the second would be used for the lift animation.

Now that I had two separate columns for each lift, I started created a function that would fill in every other column with the floor numbers. This did not end up working as intended because instead of filling in the blank column I had left for the floor numbers, the script instead generated a second canvas which it was overlaying on-top of the first set of columns on the main canvas. I later realised that I would be better off rather than using a label function to format the cell to instead use a rectangular shape and fill this with block text. From here I needed to invert the sequence in which the canvas filled in each of the floor numbers as the top floor was originally being labelled as floor zero and vice-verse.

During the process of filling all the cells I am recording a dictionary of the cells representing the moving lift and assigning each a number. This will allow me to quickly change the properties any given cell based on its location within the grid. In order to allow for the resizing of the canvas window each cell representing the lift is assigned the tag ‘lfts’ and any floor number ‘flrs’ this allows the canvas to be wiped and regenerated much faster than replacing every cell individually whenever the end-user resized the window. The next problem to solver was how to allow the text to dynamically resize relative to the size of the cells. This was done by finding the height of each cell in pixels and using floor division to find an integer number of pixels such that the height of the text was two thirds the height of the cell and would shrink and grow with the resizing of the window. For some unknown reason even though I was using floor division which should have returned an integer value I ended up also needing to use a rounding function to remove the trailing zero decimal. In later versions I will revisit this to increase the systems efficiency by reducing the number of operations. There is also a conditional statement for renaming the 0th floor to G for ground floor. Later I will include various basement levels for negative floor numbers or even allow the user to rename a cell by clicking on the text. This would be useful in various building layouts such as a rooftop pool or gym in a hotel or multiple carpark levels in a shopping mall. In terms of aesthetics I would like to have different sized columns. Currently the canvas is split in half which means there is a lot of wasted space in the cell containing the floor numbers. I would prefer to have these cells be smaller compare to those animating the lifts movement.

26/01/2020

I started today by creating two new functions based on the standard case of having a single lift serve a building of ten floors. The first of these functions would generate a list of random numbers where adjacent pairs would represent the starting floor of a passenger and the second the final destination. From here this list would be passed into the move function which would colour the starting position of the lift and switch the colour of adjected cells such that the lift appeared as though it were moving up and down between floors. Because I was using a for I in range loop to edit the colour of the cells the function would fail whenever the first number in the pair was lower than the second. This is due to the need to specify a negative step sequence when finding a decreasing range (8,2,-1). I would therefore incorporate an if else statements to determine whether the starting floor was greater than or less than the second. Later on I would also need to ensure that the random number generated would ignore any pairs generated where the numbers were identical but during testing this happened so infrequently that I decided to continue development and leave this till the end. Changed the naming scheme for keys within the dictionary to (Lift\_num, Floor) instead of the originally which was (row, columns) this made referencing each cell numerically much easier as for whatever reason the developers of tkinter decided to refer to cell by a single numerical value starting from the top left corner and counting down to the bottom of each column before moving to the next column this count also starts from one instead of zero unlike every other python module.

This table shows how I was originally creating the keys for my dictionary store.

|  |  |  |  |
| --- | --- | --- | --- |
| (0, 0) | (0, 1) | (0, 2) | (0, 3) |
| (1, 0) | (1, 1) | (1, 2) | (1, 3) |
| (2, 0) | (2, 1) | (2, 2) | (2, 3) |
| (3, 0) | (3, 1) | (3, 2) | (3, 3) |

|  |  |  |
| --- | --- | --- |
| 1 | 5 | 9 |
| 2 | 6 | 10 |
| 3 | 7 | 11 |
| 4 | 8 | 12 |

The table below represents the value stored by each of the dictionary keys above. These values are the numerical id of each individual cell which Tkinter uses to apply a function on the selected cell such as changing its colour. This shown the order in which each cell is assigned a unique id number.

By changing the keys of the dictionary to look like the ones below I am now able to move the lift up and down by looking for the lift number and the floor numbers that I wish to move it between. This greatly reduces the number of calculations needed to find the cell id number thereby increasing the efficiency of the program. Of course, all these key values will be numerical in future versions further increasing the efficiency of animating the movement of the lift in the final project.

|  |  |  |  |
| --- | --- | --- | --- |
| (1, *top\_floor*) | (2, *top\_floor*) | (3, *top\_floor*) | (4, *top\_floor*) |
| (1, 2) | (2, 2) | (3, 2) | (4, 2) |
| (1, 1) | (2, 1) | (3, 1) | (4, 1) |
| (1, *ground*) | (2, *ground*) | (3, *ground*) | (4, *ground*) |

Again I needed to invert the numbering scheme for the floors in the dictionary key tuple otherwise I would need an additional operation later on to find the difference between the max number of floors and the starting floor of the passenger in order to find the cell id number by subtracting the difference from the top floor number. By setting up the dictionary correctly before performing other functions with the dictionary I would later be able to easily reference each cell without involving additional math.

I now faced a problem with indexing as the top floor would be referred to by (1, 9) in a ten floor building counting from G or zero upwards therefore I needed to limit my range of random numbers by decrementing the number of floors by one. I decided to store the numerical value of the top floor in a separate variable as this would be referenced in multiple functions later to ensure the lift did not travel higher than the top story of the building. Doing so allows me to cut down on the number of operation as I would no longer need to subtract 1 from the ‘num\_floors’ variable each time I generated a random list or wanted to lookup the cell id of the top floor in the ‘lift\_pos’ dictionary.

I forgot to regenerate a new set of random numbers whenever a pair of matching numbers appeared in the list. This was a quick fix but should have been something I was looking out for.

I would now split the single function I had responsible for highlighting the starting floor and outputting the range of floors between the starting and end floor into two separate functions. This would allow me to highlight and unhighlight adjacent cells moving up and down the column between the passenger’s floors giving the appearance that the lift was moving up and down. I could call the second function multiple times until the lift had come to a stop. This would also make future development and improvements to the algorithm much simpler as I only required the cell ids for this function to work and could use separate algorithms for choosing which order passenger are served.

29/01/20

Found an issue with resizing the window which lead to the random lists being regenerated as this function would be called as a result of the root canvas being updated. Therefore, I needed to make this a stand-alone function. By doing so I would have an easier time later in the development process when trying to implement threading to run the lifts simultaneously as opposed to my originally idea of switching between lifts when completing each successive operation. The result of this was that the starting position of the lift flickered around whenever the window was resized.

01/02/20

I recently switched IDE’s from JetBrains’ PyCharm to Microsoft’s Visual Studio Code. Upon running my program, I realised that whenever closing out of the TKinter window by clicking the red cross button VS Code would attempt to forcibly keep the program running. To solve this issue, I implemented a function responsible for creating a popup box wherein the user could choose to quit the program or to return to the main window. Upon clicking quit the function calls window.destroy() which safely closes out of the program and stops VS Code from continuously reopening the window. Since receiving the official briefing on the project, I have changed the lift mechanism to always begin at the ground floor and run the generated sequence from there. Resizing the windows now saves the state of the program including the current position of the lift. I have updated the random generator to include the direction that the passenger wishes to travel as well as the floor that the passenger is starting on so that when the lift reaches them the destination floor is then determined.

02/02/20

Now that I had most of the UI complete, I decided to create a separate module for the algorithm responsible for running the lift. For this I would to include functions to deal with:

* Deciding which passenger should be picked-up and in which order.
* Counting the number of floors travelled for the entire sequence
* Comparison of the more efficient system with the naive system stated in the brief
* Queuing system to ensure no passenger is left waiting for a prolonged period
* Passenger exceeding the maximum capacity of the lift
* Calculating average wait time per passenger to use in the analysis of both systems

Fix and issue with resizing the window which would result in cropping the animation. This was solved by specifying a minimum window size so that the user is unable to shrink the window below this size.

09/02/20

Created and input box anchored to the bottom of the window which would later be used to specify the number of passengers used to run the sequence. This could later be expanded to include a count-down for the remaining number of passengers left to transport or implementation of a control panel including more input boxes to edit the number of floors in the building, number of lifts, as well as incorporate buttons for starting, pausing, and autocompleting the sequence. The next thing to do was to ensure that the status bar remained fixed to the bottom of the canvas. Initially I found that whenever shrinking the window the control panel would disappear behind the lift animation.

12/02/20

Started working on the logic behind the user interface. Shifted the random generator to a separate module. This new system would rely on three separate dictionaries where the key to each was the unique id of the number of the passenger. The first dictionary ‘waiting\_dict’ would use this number to access a value containing the users button choice to move either up or down within the building.

Once the lift reaches a passenger the key-value pair is moved from this first dictionary to the ‘moving\_dict’ which signifies that a passenger is inside the lift. From here the value containing up or down is appended with the floor that the passenger needs to be transported to. We can use the length of this dictionary to check whether the lift is over capacity and I will later add an exception method to stop the lift from moving when overweight. The final dictionary ‘arrived\_dict’ will store all passenger that have been transported to their desired floor along with where the passenger came from and is going to, we can once again append the value store by each key with the number of floors that the lift had to travel to serve this passenger. This can be used in later analysis to test the efficiency of the system. From here we can find ratios and averages using the total number of floors travelled by the lift compared to the combined number of floors that all passengers have moved.

14/02/20

There are two different way in which I could link the modules I have designed. One is to allow events from the user interface to effect the logic of the program of alternatively to do the reverse by which every event is triggered directly through the logic module and the user interface in acts passively and only responds to instructions given by this module with no feedback on event states.

The main instance of this that I need to consider is when moving the lift. I can either use for floors in range starting position to end position of passenger travel move the lift in a certain direction. The alternative is to have the user interface handle moving the lift and return to the logic module when done.

15/02/20

I devised an algorithm to get the lift moving. The lift starting from the ground floor check to see if anyone is waiting on this floor and if not moves up one floor and will continue following this process until it comes across a passenger. Once a passenger is collected the lift will continue to their destination floor picking up any additional passenger travelling in the same direction. If a new passengers floor occurs before the old passenger, then the lift will drop them off first and again pick up any passenger travelling in the same direction before carrying the original passenger to their destination. The process repeats over and over until all passengers are server. If the lift has reached the top of the building without picking up and passengers, then it will switch directions. This algorithm is similar to the base case but has the added benefit of being able to change directions whenever a passenger destination is determined. Later on I will change this method to be more active in determining where the lift should travel to next, for example if two people waiting on adjacent floors are travelling in the same direction the lift will later be smart enough to collect both passenger before the first is allowed to change the direction in which the lift travels. Because passengers are not arriving in real time but are instead already generated before the lift begins to move it may later be possible to perform analysis to calculate on average the most efficient route based on probabilities from multiple simulations using only the direction of travel of each passenger. We will have to see how my algorithm develops but even this standard solution is better than the base case. I may need to change my data structure to ensure that the direction of travel can be accessed quickly with the minimal number of operations. Dictionary values which are themselves a list of data may not be the most suitable for this application but ill update this log if I do decide to take a different approach.

28/02/20 – 31/03/20

Between the dates above I spent an average of two hours more or less every day rewriting my program in an Object-Oriented fashion. This increased to closer to five hours almost every day since leaving university for the year on the 17/03/20 as a result of the ongoing situation. I will link below my OneDrive with full version history of my program such that I am able to discuss each in more detail but will provide a brief overview of everything I have achieved in the past month or so.

Starting with versions 30 through 39 I did a complete rewrite of the user interface focussing on validating the users input and allowing the number of people in the building and number of floors to be changed readily using the interface instead of relying on edits to the code. The new interface consists of three columns, one for waiting passengers, one for those travelling in the lift and for passengers delivered to their desired floor. The lift operates as a series of coloured tiles filling the central column where each row splits the column into equally sized cells such that the lift colour can be changed to illustrate that the lift has arrived and departed from a floor. The leftmost column contains the floors numbers and will eventually also include as the number of people waiting on each floor. The final column will later show the number of people delivered to a particular floor.

After having created a template for the interface with most of the elements required for the final version, I started work on the logic of the lift. This is shown in the changes made between versions 40 and 49 of my system.

01/04/20 – 25/04/20

Summary of the past months work:

Versions 50 through 59 details the removal of Floor objects as a means to store waiting passengers. This was changed in favour our two simplified dictionaries, one for waiting passengers and the other for those having been delivered. Where each dictionary key value pair where the key number relates to the floor and the value store is a list containing all the people object waiting to be collected. This was far simpler in the long run and less computationally expensive then having to assign people to a list and keep track of where each list was stored and how to retrieve peoples details such as their originating and destination floor.

Versions 60 through 63 are the development stages require to include a configuration console to allow for the users input of number of passengers, floors, repeats, animation delay, and system logic. This also includes the scripting require to validate each of the users input to ensure that the main logic is able to function correctly and return and error messages to the user before starting the simulation.

Version 63 is the completed version of the basic system.

Versions 70 through 75 are the improved logic where the lift is not impeded by having to reach either end of the shaft before being allowed to change directions. These versions implement two sets, as a means to determine the next useful floor to which the lift should travel, by taking the intersection of these two sets, where if none exists the lift immediately changes directions. This because it is thereby determining that no further operations can be carries out if it were to continue.

Versions 80 through 85 are an attempt at implement a priority queuing system as a mean to further increase the efficiency of the improved system.

This leaves only version 90 through 96 as well as the completed graphical version lift.py. These versions are concerned with logical performance only such that the generation of graphs needed for the statistical analysis can occur as swiftly as possible. To this end version 96 is a version of the code incorporating both the basic and improved version of the building’s main logic such that the two can be compare side by side and plotted on the same axis.

The final version of the code lift.py is simple a combination of the two systems in one file such that it can be easily appended to the report. For this version I have also finally got around to making a functioning toggle switch to choose between both of the different lift system such that the user can select which they would prefer to see animated.