Lift System Coursework

ECM1414

Jake Phillips - 740046530
Doruk Aksu - 740087023
Thomas Wilkin-Jones - 740008888
Oliver Robson - 740012876

Generative AI Statement

This coursework has not used Generative AI anywhere in this project

Project workload breakdown

- Object Creation - Oliver
- Boilerplate code - Oliver
- JSON handling - Jake
- SCAN algorithm - Thomas
- LOOK algorithm - Thomas
- MYLIFT algorithm - Doruk
- Algorithm testing - Oliver
- Gather results - Jake
- Documentation - Jake
- Record presentation - Everyone
- Compose presentation - Jake

Links

The link to the video presentation can be found here: https://youtu.be/ngtIVcn2B1Q
This project can be found on Github here: https://github.com/jhp212/lift-coursework

Introduction

Lift systems are used all around the world. Every lift in the world needs some sort of algorithm to determine their behaviour. In this coursework, we aim to compare different lift algorithms and analyse their performance against each other. Namely, we will be looking at the SCAN algorithm, the LOOK algorithm, and our own improved algorithm, MYLIFT. We will then compare these algorithms to each other on 4 metrics:

- Average total time taken
- Average individual journey time
- Average longest journey time
- Average real time taken

The first 3 are measured in “fake seconds”. These give a representation on, if these systems were to be used in the real world, how long it would take to run. For this project we say it takes 10 “seconds” to move between floors and 2 “seconds” to open and close the doors.

SCAN Algorithm

The SCAN algorithm is a very simple algorithm and it only has a few steps:

1) Move to the next level.
2) Let off anyone who needs to get off at this level.
3) Let on as many people as possible who are waiting on that floor and who are travelling in the same direction as the lift.
4) If the lift is at the top or bottom, change direction.

Benefits

This is an incredibly simple algorithm and does not take a lot of work to implement. Furthermore, no complex calculations are needed to determine where the lift should go, making it very resource efficient to run.

Drawbacks

This algorithm is not very time efficient, as explored further in the next section.

LOOK Algorithm

The LOOK algorithm differs from the SCAN algorithm in its ability to change its movement direction at any point in its journey. It will check all the floors ahead in the current direction that it is travelling and will check if it needs to make a stop there, either to pick up or drop off any passengers. If it does not need to continue in its current direction, it will begin moving in the opposite direction.

Benefits

This algorithm is more efficient compared to the SCAN algorithm as it does not need to reach the top or bottom to change direction.

Drawbacks

There are not many drawbacks compared to the SCAN algorithm, except that the LOOK algorithm is more difficult to implement.

MYLIFT Algorithm

The MYLIFT algorithm works using a passenger priority queue strategy. It determines this priority based on the passengers start location, with respect to the lift's current position, and their total journey distance. This priority is implemented using a priority queue and sorted by using a heap sort.

For example, let's say the lift is at floor 3. A passenger is waiting on floor 6, wanting to go to floor 1. The cost will be 3 + 5 = 8, because the lift is currently 3 floors away from the passenger, and the passenger's journey is 5 floors long.

Additionally, an extra 0.1 is added to the cost per fake second that the passenger has been waiting for. A group bonus is awarded to each passenger for the number of passengers that are also waiting on their floor. This is to prioritise larger groups of passengers.

Benefits

This algorithm prioritises shorter, closer journeys first, meaning that the lift usually has fewer people in it to start with. This means that it can process more people more quickly.

Drawbacks

This algorithm will cause people whose journeys are longer, and people who are waiting further away from the lift to be deprioritised.

Theoretical analysis for time complexity - SCAN vs LOOK

Scan algorithm time complexity

For these hypothesis, we will assume that the number of passengers are proportional to the number of floors, which we will call n, and the total passengers will be $c \times n$. Additionally, let the number of passengers the lift can hold be c_{ap}.

In the best case, each of the passengers have to move only 1 floor, and form a chain going up and down (eg Floor 1 → Floor 2, Floor 2 → Floor 3, …, Floor $n - 1$ → Floor n, then Floor n → Floor $n - 1$, Floor $n - 1$ → Floor $n - 2$, …, Floor 2 → Floor 1 then back up again - if $c_{ap} > 1$, there are up to c_{ap} copies of the Floor 1 → Floor 2, up to c_{ap} copies of the Floor 2 → Floor 3 and so on.

This means that when we travel up, we pick up c_{ap} passengers on the current floor, move up one floor, drop all of the passengers that they picked up, then rinse and repeat for all the floors.

Therefore, in the best case, the algorithm runs in $O(n)$ time.

In the worst case, you have everyone on the bottom floor, and they all want to go to the bottom floor. This means all $c \times n$ passengers require n pick-up moves, and n drop-off moves per "cycle". Because we can take c_{ap} passengers per cycle, we will have $\frac{c \times n}{c_{ap}}$ cycles of $2n$ moves, which results in a total of $\frac{2c}{c_{ap}}n^2$ cycles, which is in $O(n^2)$.

Look algorithm time complexity

For these hypothesis, we will assume that the number of passengers are proportional to the number of floors, which we will call n, and the total passengers will be $c \times n$. Additionally, let the number of passengers the lift can hold be c_{ap}.

It is trivial to assume the LOOK algorithm is non-redundant (i.e. each lift movement is either heading to pick up a passenger or to drop off a passenger).

In the best case, each of the passengers have to move only 1 floor, and form a chain going up and down (eg Floor 1 → Floor 2, Floor 2 → Floor 3, …, Floor $n-1$ → Floor n, then Floor n → Floor $n-1$, Floor $n-1$ → Floor $n-2$, …, Floor 2 → Floor 1 then back up again - if $c_{ap} > 1$, there are up to c_{ap} copies of the Floor 1 → Floor 2, up to c_{ap} copies of the Floor 2 → Floor 3 and so on.

This means that when we travel up, we pick up c_{ap} passengers on the current floor, move up one floor, drop all of the passengers that they picked up, then rinse and repeat for all the floors.

This means that you travel $\frac{c \times n}{c_{ap}}$ times, so the total movement time for the lift is $\frac{c}{c_{ap}} \times n$, which is in $O(n)$.

In the average case, each passenger will have to travel on average $\frac{n}{3}$ floors. If $c_{ap} = 1$, the total necessary "drop-off" travel for each of the passengers will be $c \times n \times \frac{n}{3}$, which is $\frac{c}{3} \times n^2$. Any necessary pick-up travel will also be, on average, a third of the floors, so the total "pick-up" travel will be $c \times n \times \frac{n}{3}$. Therefore, the total travel time will be $\frac{2c}{3}n^2$, which is in $O(n^2)$.

If $c \times n > c_{ap} > 1$, it is trivial to see that this will reduce the overall number of necessary "pick-up" steps, as we do not then have to return to the floor on which the pick-up was found, and can instead focus on dropping off the passenger, thus the total minimum travel time will still be in $O(n^2)$. However, this increase may reduce the time complexity w.r.t. the number of floors, which can probably be proved using practical analysis and a log-log plot. This logically results from the following statements: if $c_{ap} \geq c \times n$, there is more capacity in the elevator than passengers in the system, so we can simply send the lift straight up, opening the doors at every floor, then straight back down and it is then **guaranteed** to have transported everyone where they need to go in $2n$ steps, which is in $O(n)$. Therefore, for $1 < c_{ap} < c \times n$, $O(n) \subset O(T(n)) \subset O(n^2)$

Finally, in the worst case, you have everyone on the bottom floor, and they all want to go to the bottom floor. This means all $c \times n$ passengers require n pick-up moves, and n drop-off moves per "cycle". Because we can take c_{ap} passengers per cycle, we will have $\frac{c \times n}{c_{ap}}$ cycles of $2n$ moves, which results in a total of $\frac{2c}{c_{ap}} n^2$ cycles, which is in $O(n^2)$.

Therefore, the LOOK algorithm will be $O(n)$ best case, $O(n^a \{1 \leq a \leq 2\})$ average case and $O(n^2)$ worst case.

Objectives

1. Implement .json files as input files
    1.1. These files should contain:
        1.1.1. The number of floors
        1.1.2. The capacity of the lift
        1.1.3. The requests at each floor
2. Implement the lift as an object.
    2.1. This object should store information about the lift's current state.
    2.2. It should also contain methods related to changing that state.
3. Implement the passengers as objects.
    3.1. These objects should store information on where passengers originate, where they want to go to, and how long they have been waiting for.
    3.2. They should also contain methods relating to changing the wait time, and updating its state.
4. Implement efficient algorithms to handle requests.
    4.1. Implement the SCAN algorithm to move the lift in one direction, picking up and dropping people off as it goes. Then, once it reaches the top / bottom, switch direction.
    4.2. Implement the LOOK algorithm which is the same as the SCAN algorithm, except that the lift can change direction at any point.
    4.3. Prioritise requests based on factors such as:

4.3.1. Direction of travel
4.3.2. Proximity to the call
4.3.3. Waiting times

4.4. Implement an additional algorithm, called MYLIFT, in an attempt to outperform the SCAN and LOOK algorithms.

5. Compare the performance of these algorithms.
   5.1. They can be compared on factors such as:
      5.1.1. Average wait time
      5.1.2. Longest wait time
      5.1.3. Algorithm run time
   5.2. To see how these factors change, we can vary:
      5.2.1. Number of floors
      5.2.2. Number of total Passengers
      5.2.3. Capacity of the lift

Design

Language

For this project we decided to use the Python programming language as all four of us are very familiar with the language. It also has built in libraries and functionality for object oriented programming, which will be heavily utilised throughout this project.

Input File

The sample input file that we are given is a .txt file. It is possible to implement this, however, we have decided to use .json files instead for a few reasons:

1) .json files are easier to parse through compared to a .txt.
2) .json files have a set structure whereas .txt files do not. .txt files have a lot more variety in the structure and can make it significantly more difficult to parse.
3) Python has a built-in json module which allows for the direct translation of .json files to python dictionaries and vice versa.
4) .json files are significantly easier to read than a text file in this format

These .json files have a similar structure as the .txt sample file:

Text file example

```
# Number of Floors, Capacity
12, 6
# Floor Requests
1: 5, 3, 8, 10, 2, 7, 9, 6, 12, 11, 4, 3, 5, 7, 9
2: 4, 1, 6, 8, 11, 12, 3, 5, 7, 9, 10
3: 2, 4, 5, 7, 9, 8, 6
4: 1, 12, 5, 3, 6, 8, 10, 2, 7, 11, 9, 6, 8, 10
5: 2, 4, 6, 7, 8, 9, 11, 12, 1
6: 10, 3, 5, 7, 8, 12, 1, 4, 2
7: 9, 2, 4, 5, 6, 8, 11, 12, 3, 10, 1, 9, 11, 12
8: 1, 3, 4, 5, 6, 7, 9, 10, 11, 12, 2
9: 2, 4, 5, 10, 6, 7, 8
10: 8, 4
11: 8, 3, 5, 4
12: 8, 9, 2, 5, 10
```

.json file example

```
{
    "floor_count":12,
    "capacity":6,
    "floor_requests": {
        "1": [5, 3, 8, 10, 2, 7, 9, 6, 12, 11, 4, 3, 5, 7, 9],
        "2": [4, 1, 6, 8, 11, 12, 3, 5, 7, 9, 10],
        "3": [2, 4, 5, 7, 9, 8, 6],
        "4": [1, 12, 5, 3, 6, 8, 10, 2, 7, 11, 9, 6, 8, 10],
        "5": [2, 4, 6, 7, 8, 9, 11, 12, 1],
        "6": [10, 3, 5, 7, 8, 12, 1, 4, 2],
        "7": [9, 2, 4, 5, 6, 8, 11, 12, 3, 10, 1, 9, 11, 12],
        "8": [1, 3, 4, 5, 6, 7, 9, 10, 11, 12, 2],
        "9": [2, 4, 5, 10, 6, 7, 8],
        "10": [8, 4],
        "11": [8, 3, 5, 4],
        "12": [8, 9, 2, 5, 10]
    }
}
```

Programming Paradigm

We decided to implement an Object Oriented programming paradigm for this project, creating a Lift class and a Passenger class. We have decided to do this for a few reasons:

1) We are able to store information about the lift and information about each passenger individually within their respective objects.
2) Each class contains methods pertaining to that class, such as the methods for operating the lift.
3) Many passenger objects can be created at once using a simple loop.

Pseudocode

Here is the pseudocode for the basic SCAN algorithm:

```
while passengers waiting:
   if lift is at top or bottom:
      flip direction

   for each passenger on board:
      if passenger.end_floor == lift.position:
         passenger.get_off()

   for each passenger waiting:
      if capacity reached:
         break
      if (passenger.start_floor == lift.position) &&
         (passenger.direction == lift.direction):
         passenger.get_on()
   move to next floor
```

This pseudocode states that the lift should move up and down, picking up and dropping people off on the way and only change direction when it reaches the top or bottom.

Here is the pseudocode for the LOOK algorithm:

```
while passengers waiting:
   if lift is at top or bottom:
      flip direction

   for each passenger on board:
      if passenger.end_floor == lift.position:
         passenger.get_off()

   for each passenger waiting:
```

```
      if capacity reached:
         break
      if (passenger.start_floor == lift.position) &&
         (passenger.direction == lift.direction):
         passenger.get_on()

   for each floor in current direction:
      if passenger waiting at floor:
         do not change direction
   if no passengers waiting and lift is empty:
      change direction

      for each passenger waiting:
            if capacity reached:
               break
            if (passenger.start_floor == lift.position) &&
               (passenger.direction == lift.direction):
               passenger.get_on()

   move to next floor
```

This code is more complicated. It operates the same way as SCAN, however once it reaches a floor, it always checks if there is anyone waiting for all floors in the lifts current direction

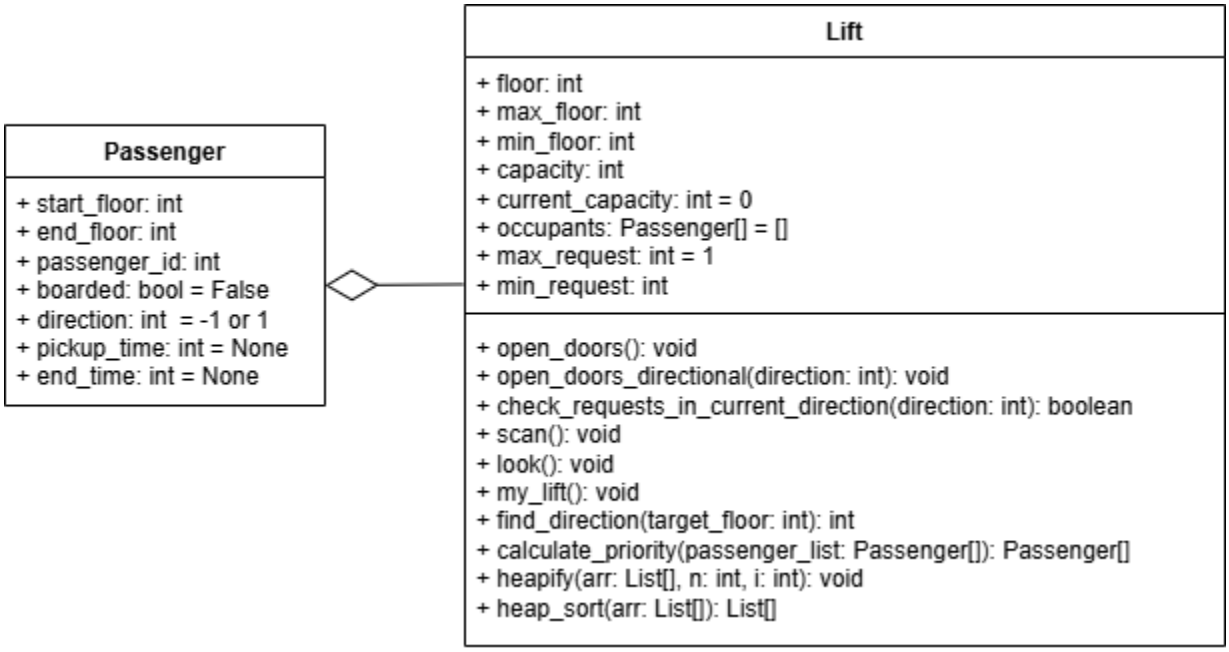
Here is the pseudocode for the MYLIFT algorithm:

```
while passengers waiting:
   if lift is at top or bottom:
      flip direction
   priority_queue = calculate_priority(passenger_list)
   current_passenger = priority_queue[0]
   while lift.position != current_passenger.start_floor:
      move to next floor
   let on passengers
   while lift.position != current_passenger.end_floor:
      move to next floor
   let off passengers
```

This may look simpler than SCAN and LOOK, however to calculate the priority of each passenger will take more effort to set up the list and then sort it using a heap sort.

Technical Implementation

UML Class Diagram



Above is the UML Class Diagram for the Passenger and Lift classes. The Passenger class is aggregated with the Lift class. The Lift class contains a list of Passenger objects.

Passenger Class

The passenger class contains 8 attributes:

- start_floor: an integer that indicates the starting floor of the passenger.
- end_floor: an integer that indicates where the passenger wants to go.
- passenger_id: a unique integer that can identify a passenger.
- boarded: a boolean that indicates whether a passenger is inside of a lift. Defaults to False.
- direction: an integer that indicates which direction the passenger wants to go. -1 for down, 1 for up.
- pickup_time: an integer to tell at what time the passenger was picked up. Defaults to None.
- end_time: an integer which holds the time that the passenger was picked up. Defaults to None.

Lift Class

The lift class contains 9 attributes:

- floor: an integer that holds the current floor the lift is on.
- max_floor: an integer that holds the highest floor of the lift.
- min_floor: an integer that holds the lowest floor of the lift.
- capacity: an integer that defines how many passengers are allowed on the lift at any one time.
- current_capacity: an integer that holds how many people are currently on the lift. Defaults to 0.
- occupants: a list of Passenger objects that contains every passenger that is on board the lift.
- max_request: an integer that contains the highest floor that a passenger is currently waiting at. This is used for the LOOK algorithm. Defaults to 1.
- min_request: an integer that contains the lowest floor that a passenger is currently waiting at. This is used for the LOOK algorithm.

The lift class also contains 9 methods:

- open_doors: lets off any passenger on board who needs to get off of the lift and lets on any passengers waiting on the current floor, up until capacity has been reached.
- open_doors_directional: lets off any passenger on board who needs to get off of the lift and lets on any passengers waiting on the current floor, up until capacity has been reached. These passengers must be travelling in the same direction as the lift to be able to get on. Takes the current direction being travelled as input. -1 for down, 1 for up.
- check_requests_in_current_direction: will return True if the lift is empty and there is a passenger waiting to board the lift in the current direction it is travelling in. It will return False otherwise. Takes the lift's current direction as input.
- scan: performs the SCAN algorithm on the current state of the lift.
- look: performs the LOOK algorithm on the current state of the lift.
- my_lift: performs the MYLIFT algorithm on the current state of the lift.
- find_direction: will find the direction the lift needs to travel in order to get to a specific floor. Returns -1 for down, 1 for up, or 0 if the destination floor is the current floor that the lift is on. Takes the destination floor as input.
- calculate_priority: for each passenger, a cost is given to them based on their distance from their destination floor. Takes a passenger list as input.
- heapify: will heapify an array into a max heap. Takes the array to be heapified, the size of the heap n, and the index of the current node i as inputs.
- heap_sort: will perform a heap sort on an array. Takes the array to be sorted as input.

File Structure

```
{ID}
├── presentation
│   └── presentation_link.txt
├── results
│   ├── charts
│   │   ├── Time Taken for varying Capacity - SCAN & LOOK.svg
│   │   ├── Time Taken for varying Capacity - MYLIFT.svg
│   │   ├── Time Taken for varying Floor Count - SCAN & LOOK.svg
│   │   ├── Time Taken for varying Floor Count - MYLIFT.svg
│   │   ├── Time Taken for varying Passenger Count - SCAN & LOOK.svg
│   │   └── Time Taken for varying Passenger Count - MYLIFT.svg
│   ├── data
│   │   ├── Capacity.json
│   │   ├── Floor Count.json
│   │   └── Passenger Count.json
├── sources
│   ├── tests
│   │   ├── curated tests
│   │   │   ├── test1.json
│   │   │   ├── test2.json
│   │   │   └── test3.json
│   ├── bulk_testmaker.py
│   ├── loader.py
│   ├── main.py
│   └── testmaker.py
├── specification
│   └── coursework_spec.pdf
├── README.md
├── requirements.txt
└── run.bat
```

bulk_testmaker.py

This file contains functions to facilitate the creation of test files and has 3 functions

- generate_test_dict: Generates a test file with the capacity, floor count and passenger count taken as input. This uses a random distribution of requests.
- save_dict_as_json: Takes a dictionary and a file path as input and will write the dictionary to the json file.
- generate_tests: Will generate all the test files necessary to generate the graphs.

loader.py

This file contains the load_file function to load a json test file, extract the data from it, and it will return the floor count, the capacity of the lift, and a lift of passenger objects. This file also contains the Passenger class.

testmaker.py

This file contains a more in depth test file generator. It functions similarly to the test generator in the main.py file, however you are able to specify if a specific floor should be weighted more heavily than other floors. It also has the capability to generate the best case scenario as a test file.

main.py

This file contains the main user interface for the program and this is the file that will be run by the user, either directly or by using run.bat.

When the user runs this file they are presented with a welcome screen and are given 4 choices:

a) Run a specific test file
b) Generate a test file
c) Test all algorithms over a range of tests and view graphs
x) Exit the program

Selecting option a) asks the user to select a test file to run. If a GUI is available, the program uses tkinter's filedialog function to ask the user. If a GUI is not available, the program asks the user to enter the path manually. It then asks the user what algorithm they want to run, either:

a) SCAN
b) LOOK
c) MYLIFT

The program will then run the test file and output the result using the print_results function which takes the results of the tests and prints it out in a more human readable format. It will then bring you back to the main menu.

Selecting option b) will ask you for the number of floors, the capacity of the lift, and the total number of people in the lift system. It will then ask you where you want to save the file. If a GUI is not present, it will ask you for the file path to save to. It will then generate the test file using the generate_test_dict and save_dict_as_json functions from bulk_testmaker.py.

Selecting option c) will then ask the user if they want to:

a) Test all of the algorithms over the range of tests
b) Generate all of the test files required

Selecting the test option will then ask you whether you want to:

a) Analyse varying capacity
b) Analyse varying floor count
c) Analyse varying passenger count

It will then ask for confirmation to run the tests and if confirmed, will run all of the test files. It will then show you the graph for the data (if a GUI is available), and save the graph as an svg file and save the raw data as a .json file.

Selecting the option to generate the test files will show a warning to the user that over 2400 files will be generated in the sources/tests folder and asks for confirmation on whether to start. If confirmed, the files will be generated. These files will have the file name:
`{parameter being varied}_{value of parameter}_{index from 0-9}.json`

The main.py file also contains the Lift class. The file also contains the run_test_file function which takes a file path and an algorithm name as input. It will then run the test file using the algorithm and return the data from it.

main.py also contains the run_range_of_tests function which takes the range of parameter values (e.g range(1,51)), the pattern for the file name, and the name of the parameter as inputs. It will then process all of the test files with all of the algorithms, generate the graphs using matplotlib, shows the graph to the user (if a GUI is available), and saves the graph as an svg and the raw data as a .json.

Results

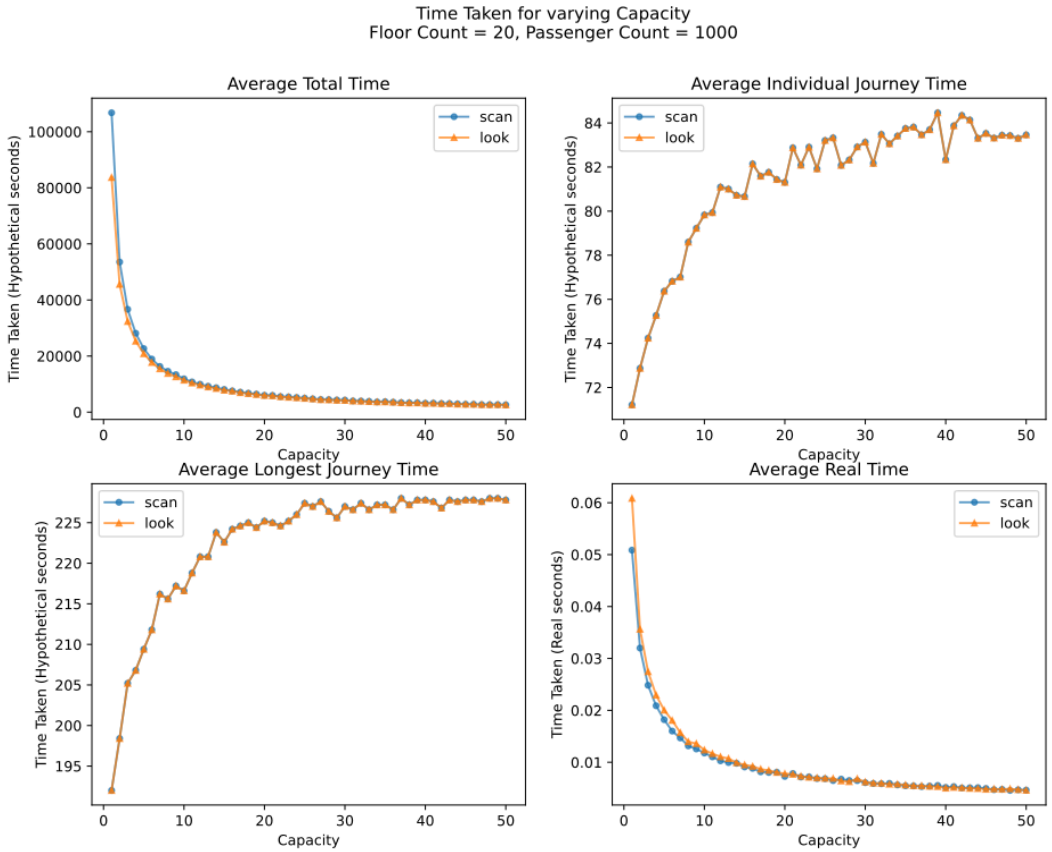
Scenarios

For this, we decided to test how the performance of the algorithms change when one of the parameters changes over a range of values. We decided to range:

- Capacity from 1-50. Floor count = 20 and passenger count = 1000.
- Floor count from 2-100. Capacity = 5 and passenger count = 1000.
- Passenger count from 10-1000. Capacity = 5 and floor count = 20.

For all of these charts, the full .svg file can be found in the project files under `results/charts`. The raw data that created these charts can be found under `results/data`.

Varying Capacity



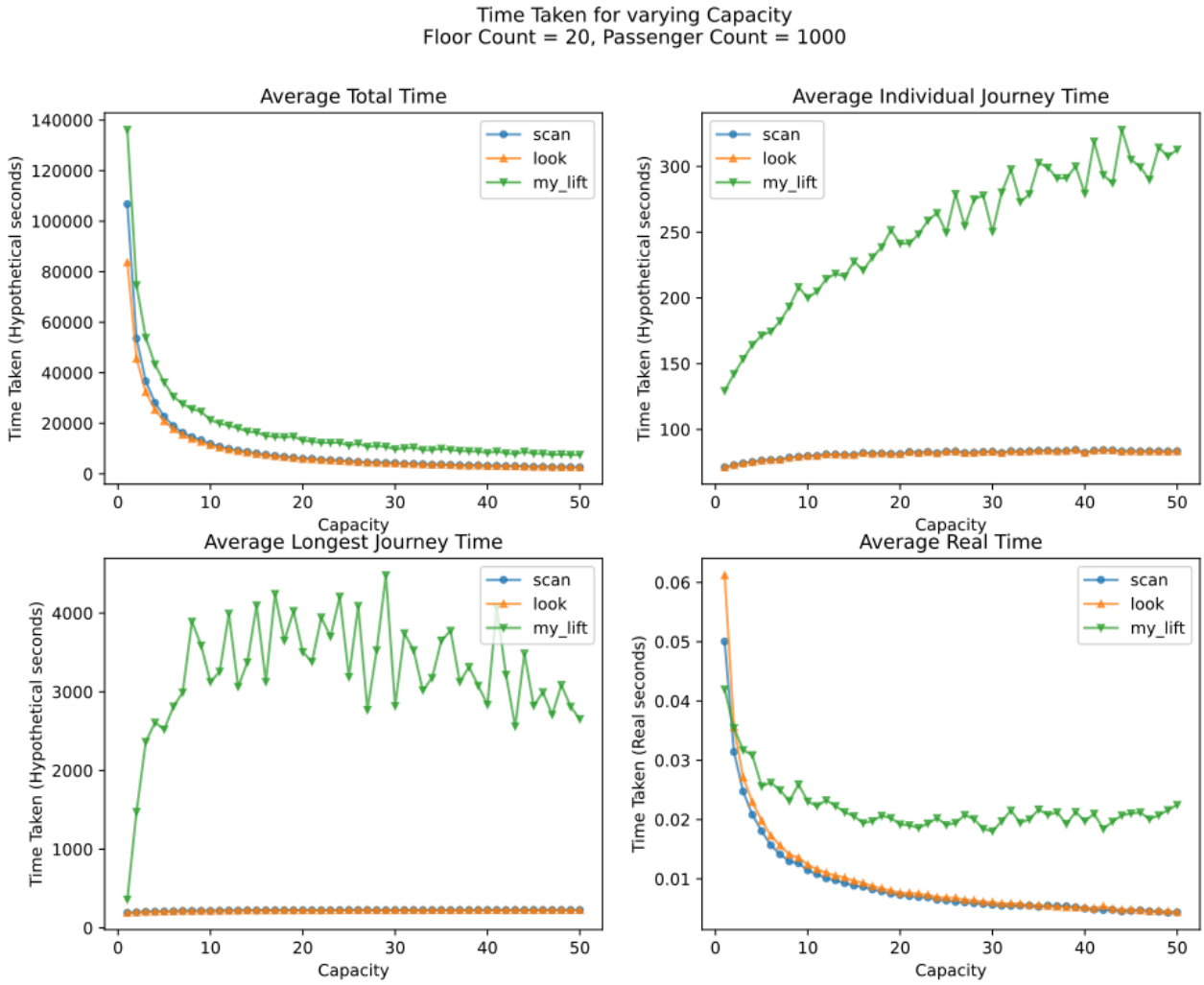
For SCAN and LOOK, we see that the average total time and average real time are both inversely proportional to capacity. This makes sense as with a higher capacity, more passengers can fit onto the lift at any one time, meaning more passengers can get to their destination at any one time.

We also see that the average individual journey time and the average longest journey increase as capacity increases. This also makes sense as if there are more people on the lift, passengers will be getting off on the way to your stop, so your time on the lift will increase.

Interestingly, the average individual and longest journeys are identical for the SCAN and LOOK algorithms at every capacity. This is because once a passenger is inside the lift, the journey that they take using both algorithms is the same as LOOK will only turn around if the lift is empty.

This should mean that the total time is lower for LOOK and indeed that is true. For very low capacities this is very noticeable where, at capacity = 1, LOOK is 9000 fake seconds faster than SCAN. However for very high capacities, LOOK is only around 10-20 fake seconds faster. This is still about 7% faster, so it is worth implementing over SCAN.

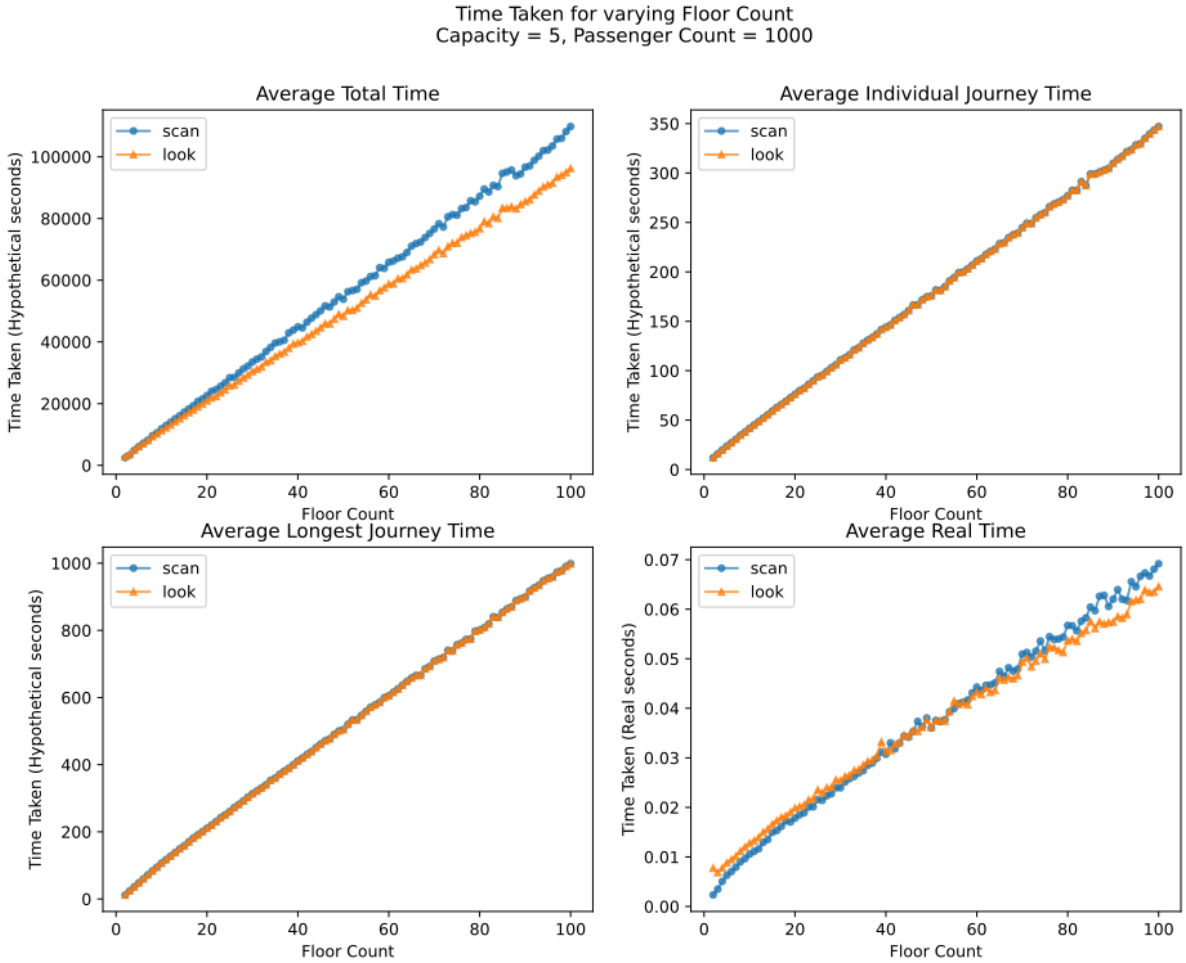
In terms of real time, LOOK seems to run slower than SCAN due to more iterations through the passenger list in order to determine if the lift should change direction.



Comparing these two to the MYLIFT algorithm, we see that MYLIFT seems to perform worse in every metric we measured. This may be due to several factors:

1) The MYLIFT algorithm only concerns itself with one passenger at a time, meaning that for the other passengers on board with that passenger, their destination is not being considered fully. This can be most easily seen in the average longest journey time.
2) Passengers who are further away from the lift, and passengers who's journeys are longer in distance, are deprioritised compared to passengers with shorter journeys. This would most likely explain the average individual journey time being higher.
3) The time it takes to calculate all of the costs for each passenger and then heap sort these values will explain why the average real time is significantly higher than for SCAN and LOOK.

Varying Floor Count

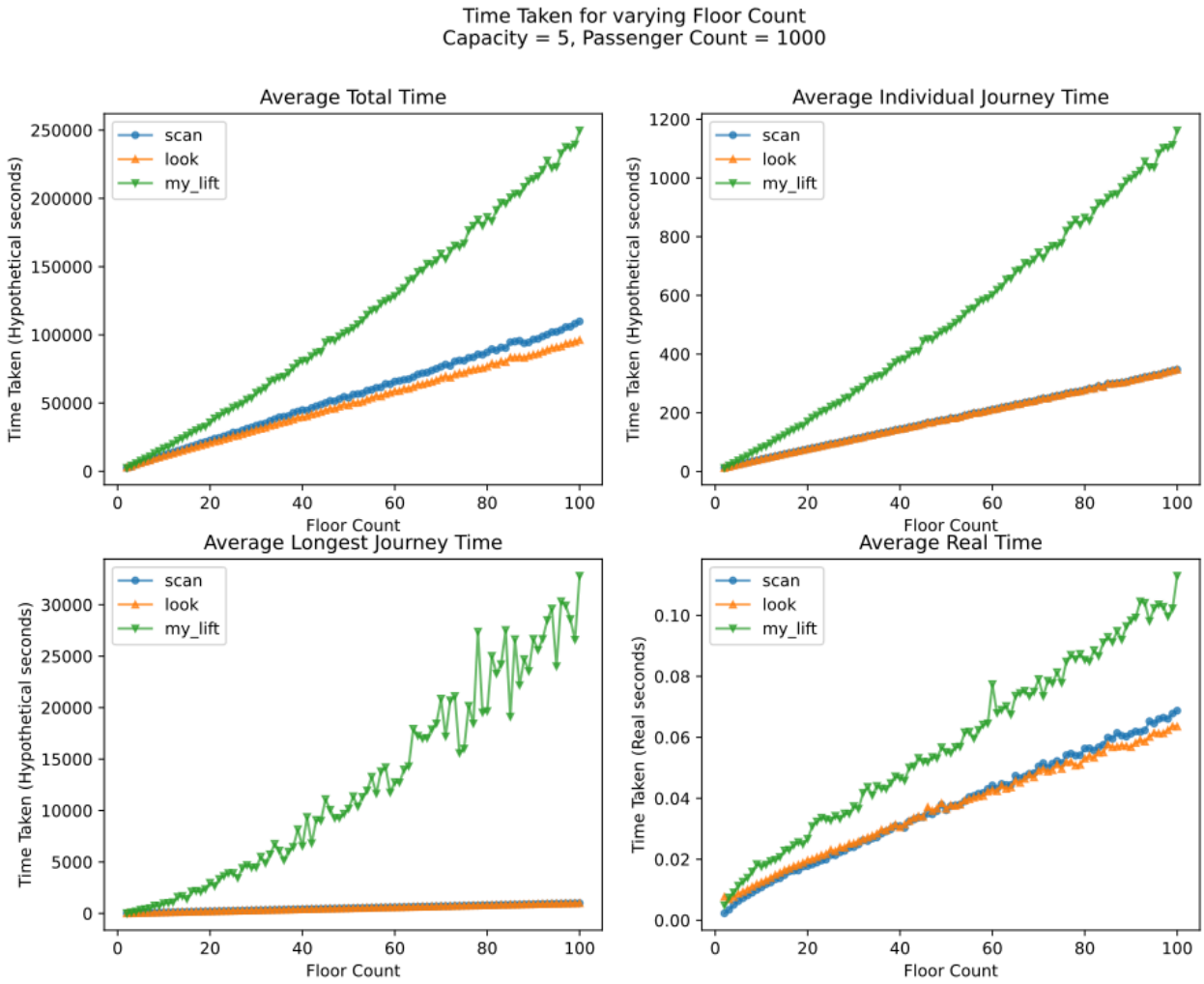


For this one, all 4 metrics measured have a positive linear relationship to the SCAN and LOOK algorithms.

The average total time for SCAN has a steeper gradient compared to LOOK, meaning that for small lift counts, the difference is smaller, with LOOK out performing SCAN by 100-300 fake seconds. However once we reach very high floor counts, the effect that the LOOK algorithm's ability to switch direction at any point has is increased, with LOOK outperforming SCAN by around 10000-12000 fake seconds.

Similarly to when we varied lift capacity, the average individual journey time and the average longest journey times for SCAN are exactly identical to that of LOOK for the same reasoning as before.

The average real time taken is interesting as for low floor counts, LOOK takes longer than SCAN for the same reason as before, however for higher floor counts, the SCAN algorithm needs to check if a passenger needs to get off for all of the floors that LOOK skips, which ends up having more of an effect, and so it takes longer to run.

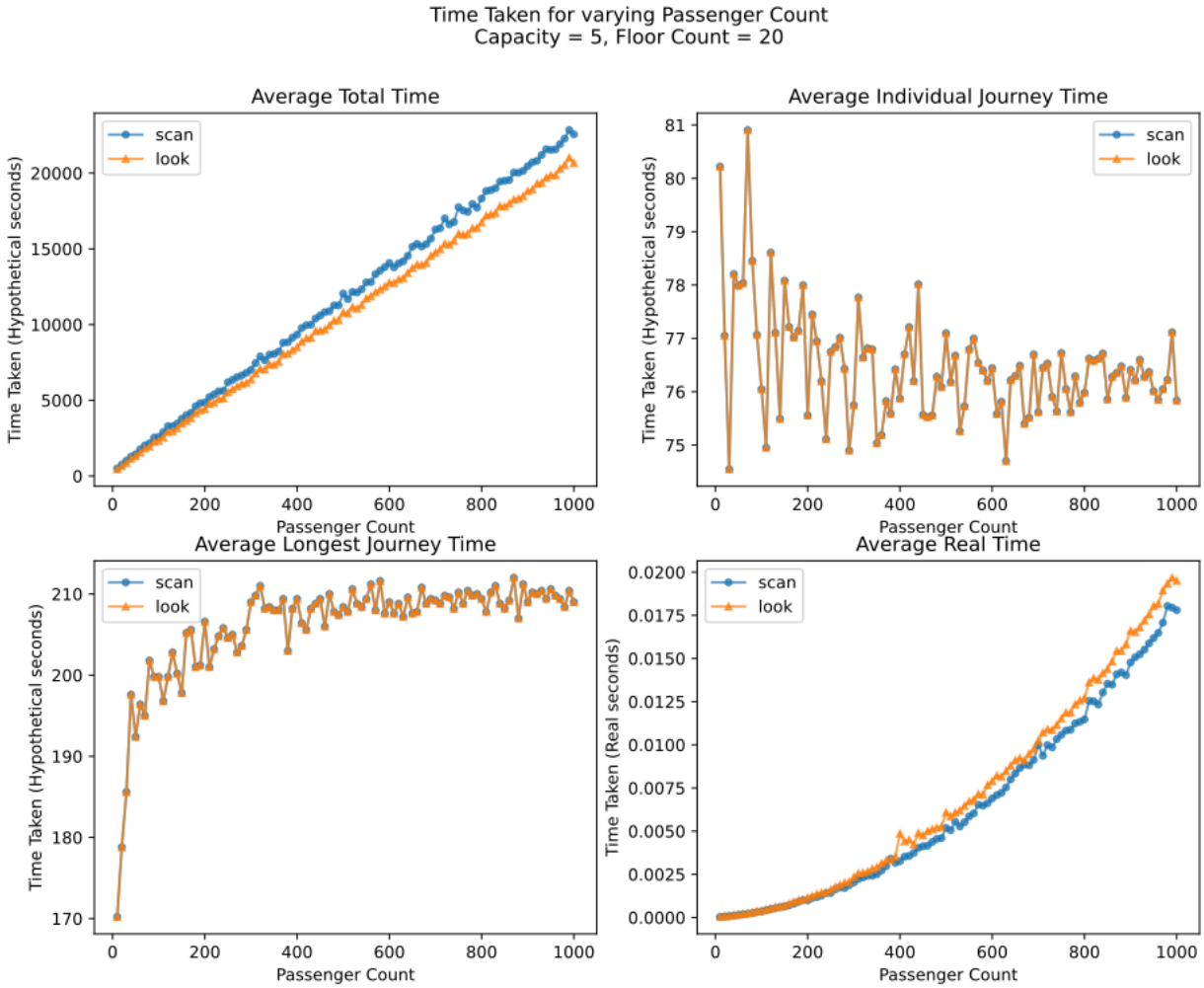


When varying the floor count, the MYLIFT algorithm seems to exhibit the same linear trends as the SCAN and LOOK algorithms. However, it is still significantly slower and less efficient than the other two for similar reasons as before.

In the same way as when we varied capacity, the average individual journey time and the average longest journey are substantially worse with MYLIFT, whereas the average total time is worse by a smaller margin.

For the average real time, again, the time to calculate and sort costs will cause this function to perform slower. However this difference is very small. For example the MYLIFT algorithm only takes around 0.05s more compared to SCAN and LOOK at 100 floor count.

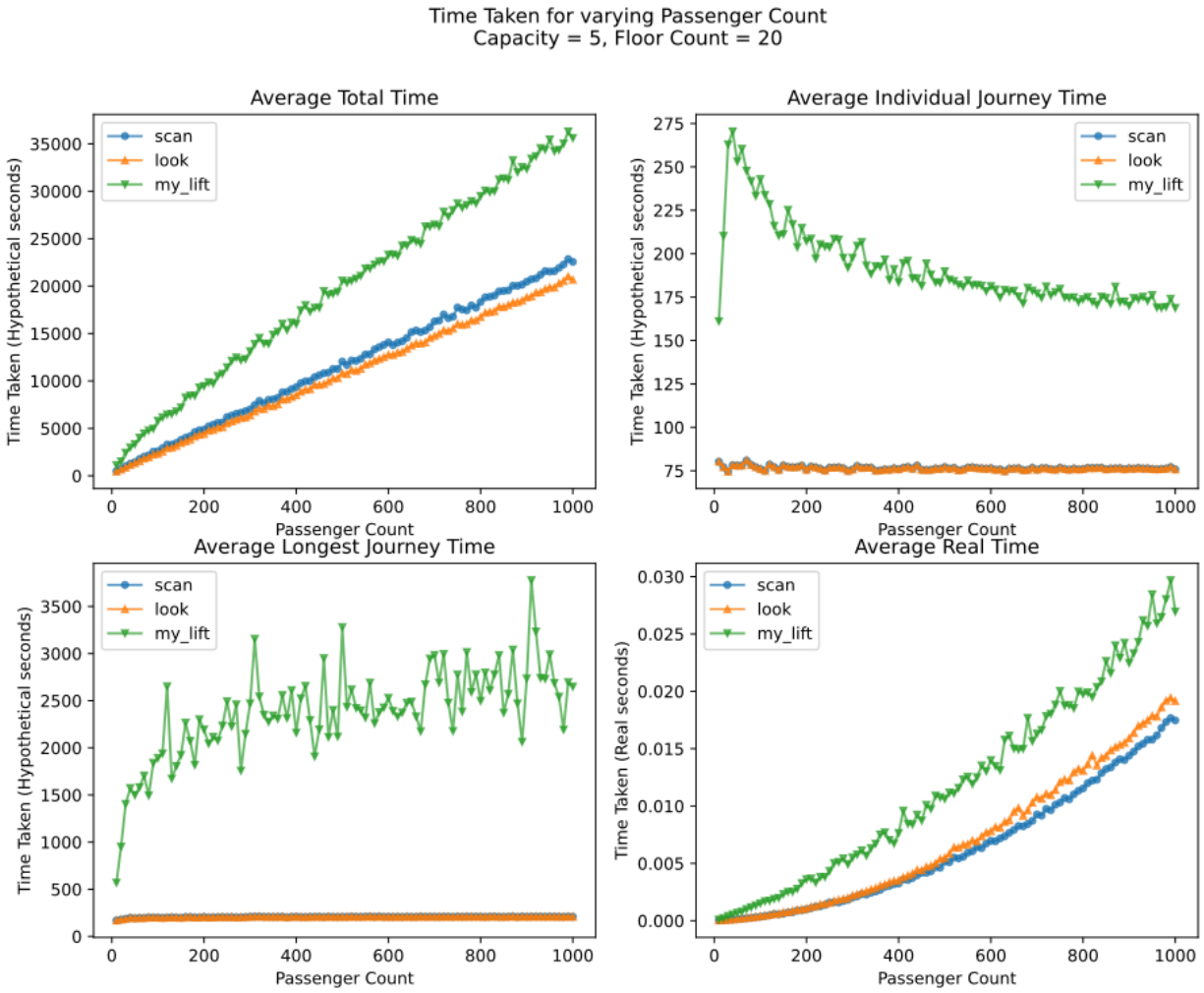
Varying Passenger Count



The average total time changes in the same way as when the floor count changed for SCAN and LOOK. The average longest journey time changes in the same way as when the capacity of the lift was changed for SCAN and LOOK.

The average individual journey time looks very random and chaotic. However these values only vary between around 74 and 81 fake seconds, with the graph seeming to stabilise as the passenger count increases. This is due to us taking the mean of all of the journey times. Since there are more passengers, there is less variety in the values, leading to less variety in the mean. The randomness seems to be mostly due to the random distribution of the passengers changing. Therefore we can conclude that changing the passenger count does not have an effect on the individual journey time for the passengers.

The average real time for both SCAN and LOOK seems to exponentially increase with passenger count, with LOOK being slower than SCAN in most scenarios. This is most likely due to LOOK having to iterate over the passenger list another time in order to check if there are any people waiting ahead of the lift.



Once again, the MYLIFT algorithm does perform worse compared to SCAN and LOOK but it also exhibits the same trends as the other two. The biggest performance hits are definitely the average individual journey time and the average longest journey time. The difference in average total time and average real time are not as pronounced.

Conclusion

In conclusion, we succeeded in completing all of our set objectives in this project. Working as a team, we implemented 3 different lift algorithms and compared them to each other. Unfortunately, we were unable to compete with the SCAN and LOOK algorithms performances with our MYLIFT algorithm.

Interpretation of results

From our testing, we determined that the LOOK algorithm is probably the best one to implement in the real world. It performs significantly better than the SCAN algorithm due to its ability to change direction at any point rather than just at the top and bottom. SCAN is also a very good option, especially for smaller lift systems as the performance difference between SCAN and LOOK at these sizes is not very much.

The MYLIFT algorithm attempts to process the passengers with the least cost to the system. Although in our implementation this algorithm performed worse compared to SCAN and LOOK, we think that this concept can be used to outperform the other two algorithms.

Ultimately, both the SCAN and LOOK algorithms are also the most ‘fair’ implementations in any real-world scenario, as they do not assign any priority to any individual; if the lift happens to pass their floor, they will be able to embark, provided the lift is not at maximum capacity. This would not necessarily prove to be the most efficient algorithm, but it does eliminate any prejudice in the system against those with longer journey times.

Future Work

If we were to work on this more, the first thing we would do is to improve the MYLIFT algorithm to try and compete with SCAN and LOOK. This could be done by possibly considering the cost of other people on the same floor as other passengers. If someone with a small cost is waiting with a lot of people with high costs then that person should also have a higher cost.