CSE 374

Algorithms

Elevator Algorithm Project Report - Spring 2022

An Algorithm to Support Buildings with Priority Floors

Seerat Boparai
Sophia Yarwick

Vaskar Raychoudhury

Abstract

Elevators are used everywhere in the modern day, but there is the issue of how they can best serve the people who use them. Every elevator has different limits and each building has unique challenges to consider. The LOOK disk scheduling algorithm offers a solution to this scheduling problem. By implementing a modified version, we hope to show that elevators can work in buildings with certain priority floors and that they still can be implemented in ways that serves the most amount of people quickly.

Contents

1 Introduction

In today's modern day and age, it's unlikely to come across a building that doesn't use elevators. As the buildings get higher and the number of people that the buildings serve gets higher, the demand for highly efficient elevators continues to rise. The goal is to minimize waiting time for each individual and only have them travel in one direction towards their destination. When huge number of floors come into play, this becomes more and more difficult.

Early elevators can be traced back to at least the third century B.C.E.[4]. People would use animals, water wheels, or their own strength to operate these elevators, which were more like hoists [1]. Other elevators used a kind of rope and pulley system to move up and down. Modern elevators came around in the 1800's and used steam power to be operated. These elevators were used mostly for lifting materials in factories and other industrial workplaces. However, they were generally unsafe for many people to use. To fix this problem, Elisha Otis created the first safety features for the elevator[SITE]. Otis created a backup to prevent the elevator from falling if the support cable broke. After these were implemented, elevators became more common as they became safer. By 1880, the first electric elevator was created, and by the 1950's, elevators started using electric buttons to control their movement[1].

According to Popular Mechanics, the simplest elevator algorithm is only two steps: "As long as there's someone inside or ahead of the elevator who wants to go in the current direction, keep heading in that direction [and] Once the elevator has exhausted the requests in its current direction, switch directions if there's a request in the other direction. Otherwise, stop and wait for a call"[2]. This algorithm is a form of a disk scheduling algorithm.

Originally used for traversing disk drives, the elevator algorithm manipulates them to use in a vertical path with two-location requests. Disk scheduling is used for I/O requests coming to the disk. There are many implementations of disk scheduling algorithms, but today, we will focus on the LOOK algorithm.

In our report, we will be implementing a version of the LOOK algorithm with priority. The elevator will run with LOOK implemented, but certain floors can have priority. If a request has either a start or destination floor that is a priority, the elevator will break the LOOK algorithm and service the priority request first. While this will reduce optimization and increase average service time, We believe his implementation is an important feature and can be seen in many buildings. For example, in a large company building with CEO, CFO, and CTO offices, or large apartment

building with special celebrity residents, elevators may have to run with this implementation. We hope to show how this modified LOOK algorithm works and show it's implementation in many cases, as well as analyze the cost and performance of this beautiful algorithm.

2 Algorithm Details

The SCAN algorithm is a common elevator algorithm that we will be referencing in our algorithm. SCAN is a Disk Scheduling Algorithm that was used to determine the motion of the disk arm and the order of completing read and write requests [6]. The advantages of SCAN are higher throughput and lower average response times. SCAN is also called the elevator algorithm because it works in the same way modern elevators do, moving in one direction and only servicing requests that are along that path in the same direction. The disadvantages to the SCAN algorithm come from long wait time for certain requests [6]. Specifically, requests either far away from the start point, or that originate from locations just visited by the elevator will have to wait longer to be serviced.

To help optimize SCAN, an algorithm called LOOK was developed to minimize these wait times. Like SCAN, LOOK only services requests along it's path, but instead of having to traverse all the way to the end of it's path, it can look ahead and see where the requests are and turn around when it reaches it's last request in any given direction [5]. For example, if an elevator services a 10-story building and someone on floor 6 requests to go to floor 9, an elevator based off the LOOK algorithm would be able to stop and turn around at the 9th floor instead of the 10th, like it would have with the SCAN algorithm.

We will use SCAN and LOOK as a basis for our algorithm. Since neither deal with priority, we will be taking the concept of SCAN and LOOK to start our elevator the way we want. From there, we will be changing and adding to the steps of these algorithms to solve the problems we want to.

2.1 Problem Definition

Elevators have been around for centuries, but automated elevators are fairly new to us. With their invention, elevators need to have a way to service the whole building as efficiently as possible. By optimizing the way the elevator moves, people will have to wait less and get around easier. To create a solution to this problem, we must create an algorithm to satisfy all of our requests.

Assumptions For our algorithm, we are assuming the following:

1. No new requests can be added one the algorithm is running. Meaning, this is not a live algorithm.
2. All requests are made in a short time span one after the other.
3. The requests are added in FIFO order but might be completed in a different order based on priorities and directions.
4. Going from one floor to another takes t units of time, which can be changed as needed.

Inputs

- **Requests(R)**: an array of Request objects. Each Request object contains:
  * $r.floor$: tells which floor the request is made from
  * $r.isPriority$: boolean that states if the request is a priority of not.
- **Elevator(E)**: an Elevator object. Each elevator object contains:
  * $e.c$: the floor (or location) that the elevator is currently on.
  * $e.d$: the direction the elevator is moving and represented as a boolean with true ("up") or false("down").

Outputs This algorithm outputs some feedback on what it is currently completing, but the main output is the time that the elevator takes to complete all requests. We assume that every time the elevator's location changes or stops at a floor, the time will increase by a specific amount.

2.2 Pseudocode

Variable Table	
Name	Amount
req {{}}	{{int, boolean}{int, boolean}...}
elevator{}	{int, boolean}

Algorithm 1 Pseudocode

```
1: Input Requests (R), Elevator (E)
2: procedure ElevatorTime(R, E)
3:     Q ← R[0]
4:     if Q = ∅ then
5:         return;
6:     while Q ≠ ∅ do
7:         if r.floor > e.c then
8:             e.d ← true
9:         else
10:            e.d ← false
11:        end if
12:        if r.isPriority then
13:            e.c ← r.floor
14:            Remove r from Q
15:        else
16:            if e.d = true then
17:                Q ← R for any r'.floor > r.floor and r'.floor < e.c
18:            else
19:                Q ← R for any r'.floor < r.floor and r'.floor > e.c
20:            end if
21:            while e.c ≠ r.floor do
22:                if Q contains e.c then
23:                    Stop at c
24:                    remove c from Q
25:                    if e.d = true then
26:                        c++
27:                    else
28:                        c–
29:                    end if
30:                end if
31:            end while
32:            if e.d = true then
33:                if r' ∈ R where r'.floor > e.c then
34:                    Q ← R any r'.floor > e.c
35:                else
36:                    e.d ← false
37:                    Q ← R[0]
38:                end if
39:            else
40:                if r' ∈ R where r'.floor < e.c then
41:                    Q ← R any r'.floor < e.c
42:                else
43:                    e.d ← true
44:                    Q ← R[0]
45:                end if
46:            end if
47:        end if
48:        end while
```

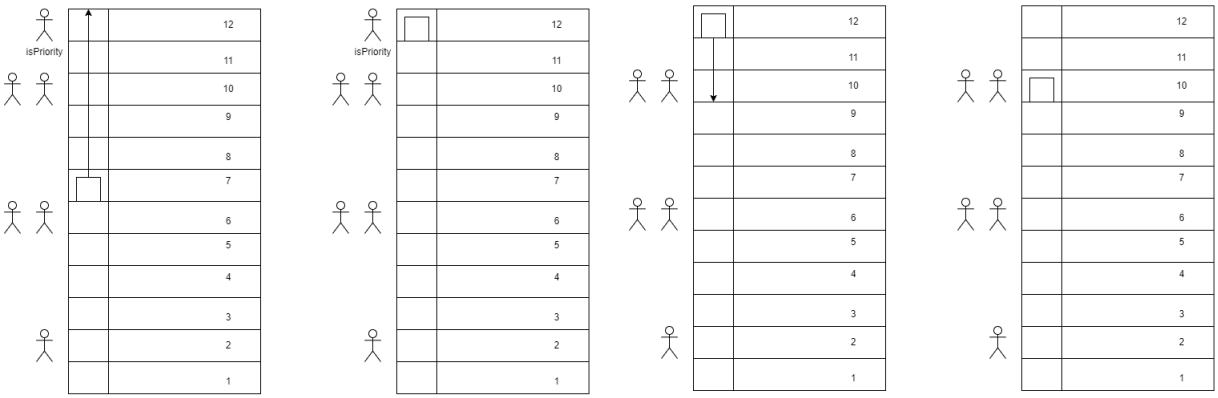
3 Use Cases

3.1 Case 1: Simplest Elevator Case

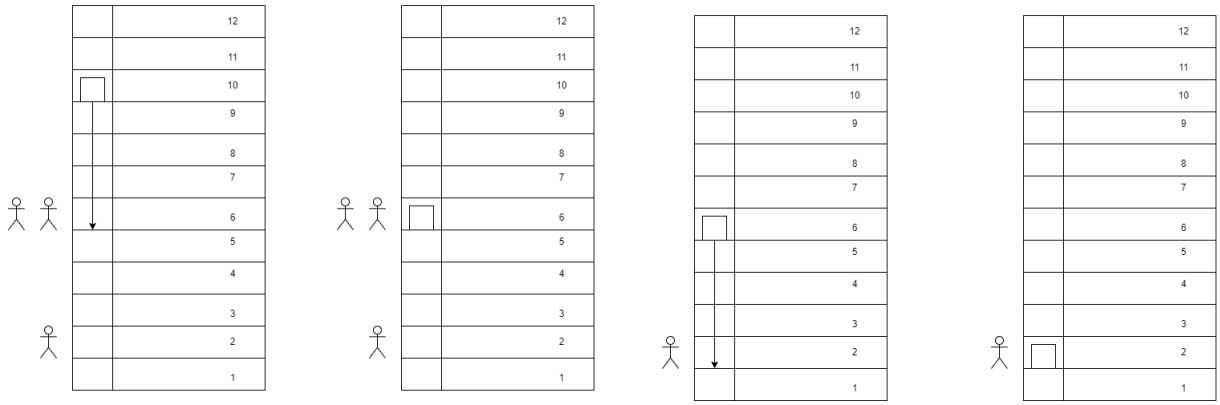
Variable Table 1	
Name	Amount
req {{}}	{{2, false}},
elevator{}	{1, up},

1. Adds {2, false} to activeReqs queue
2. While a request is made
    a) check for other requests between floors 1, 2, none
    b) elevator moves "up" and opens door on 2nd floor, request complete
    c) check for more requests in same direction, none
        i. elevator switches directions, check request in "down" direction, none
3. elevator stops at floor 2

3.2 Case 2: More Complex Elevator Case



Requests {12,2,6,10,10,6} are called. Elevator takes in first request {12} and goes immediately there because it has priority. Elevator completes request and changes direction to down. Elevator picks up {10,10}.



Elevator continues in "down" direction because more requests. Elevator stops at 6 and picks up requests {6,6}. Elevator continues down to 2 and no more requests, done.

3.3 Case 3

Variable Table 2	
Name	Amount
req {{}}	{{2, false}, {5, false}, {3, false}}
elevator{}	{1, up},

1. Adds {2, false} to activeReqs queue
2. While a request is made
    a) check for other requests between floors 1, 2, none
    b) elevator moves "up" and opens door on 2nd floor, request complete
    c) check for more requests in same direction, finds {3, false}, {5, false}
    d) elevator moves "up" and stops on 3rd and 5th floor, requests complete
    e) check for more requests in same direction, none
        i. elevator switches directions, check request in "down" direction, none
3. elevator stops at floor 5

4 Results

4.1 Cost Analysis

The time complexity and cost of the algorithm varies with these factors:

1. r : amount of requests
2. f : number of floors visited
3. p : if priority floors are called

The elevator's movement will always be less than or equal to twice the amount of floors f, since it doesn't always visit every floor. So, worst case for the elevator to reach all people and service all floors would be $O(N^2)$. Then, the average case for the elevator algorithm would be $O(N * logN)$, where N is the number of floors. We get this because the elevator must pass or visit most of the floors in one direction (N), and has to revisit some of the floors in the other direction ($logN$). Therefore, on average, the priority elevator has the same time complexity as the normal implementation.

While working through this problem, we realized this algorithm will be more costly than the SCAN algorithm it is based off of. This is because SCAN considers all requests with the same priority, deciding to go to the closest one in the direction it's currently moving. This keeps moving cost down by going to the closest request and servicing all requests in that direction. On the other hand, our algorithm considers priority, and will skip over any requests between the start location and the priority request. While this increases the time complexity and cost, we believe it was a worthy problem to consider because of it's real-life uses.

4.2 Running the Algorithm

We created a class for the Requests and Elevator, where both had methods to create the objects and get their inputs. Then, we created a main method in a tester class that created requests, added them to an ArrayList, created a single elevator object, then called the ElevatorTime() method where the total run time of the elevator would be calculated. We used both Replit IDE and Eclipse IDE as our environment for running the algorithm.

4.3 Dataset/Data source

Because of how the code is set up for this problem, our dataset is a collection of Requests objects put into a ListArray. This is passed from the main method to the Elevator class' method ElevatorTime. We created multiple different sets of data from one request (in our simplest case) to more complex cases that included many more requests.

```
List<Request> requests = new ArrayList<>();

Request Bob = new Request(2, false);
requests.add(Bob);
Request rita = new Request(5, false);
requests.add(rita);
Request tom = new Request(3, false);
requests.add(tom);
```

Figure 1: Example of creating and adding requests

4.4 Graphs/Figures/Tables

```
Elevator location: 10 | Elevator direction up/down (true/false): true
Starting Elevator...

request for floor 9 has been completed
request for floor 7 has been completed
request for floor 6 has been completed
request for floor 5 has been completed
request for floor 3 has been completed
request for floor 2 has been completed
Requests finished

Completed in 590300 nanoseconds
```

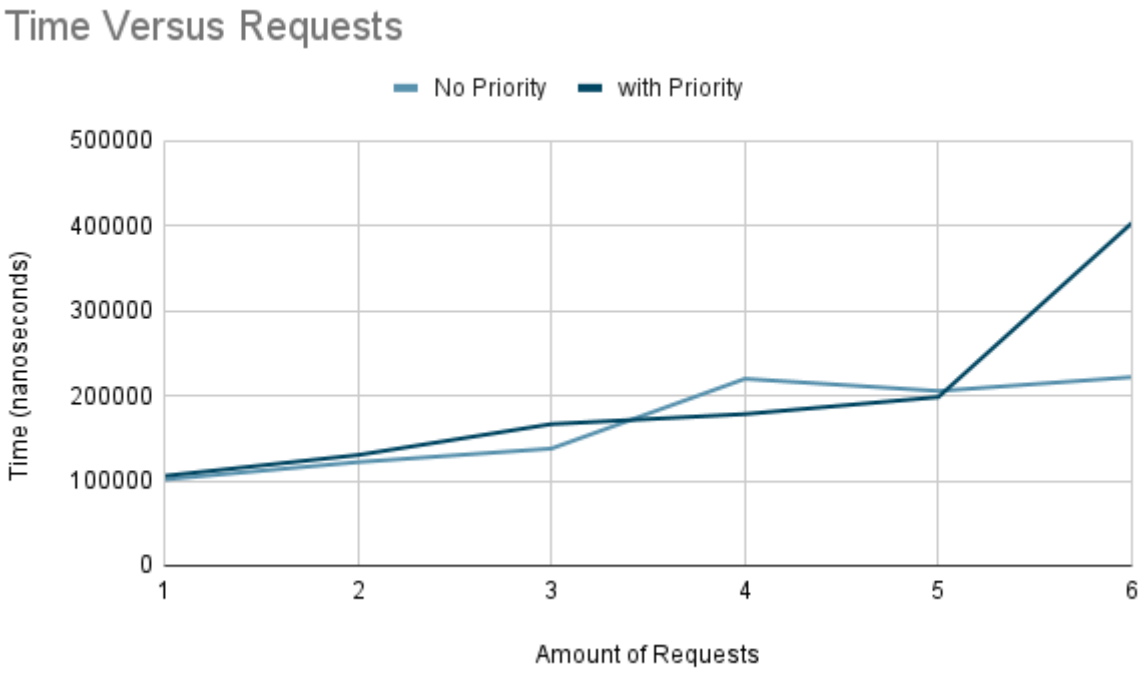
Figure 2: Simple Case. Requests: (2, false), (5, false), (3, false), (7, false), (6, false),(9, false)

```
Elevator location: 5 | Elevator direction up/down (true/false): true
Starting Elevator...

request for floor 3 has been completed
request for floor 2 has been completed
Switching directions
request for floor 5 has been completed
request for floor 6 has been completed
request for floor 7 has been completed
request for floor 9 has been completed
Requests finished

Completed in 663100 nanoseconds
```

Figure 3: Case with Priority

Requests: (2, false), (5, false), (3, **true**), (7, false), (6, false),(9, false)

Requests vs Time		
Amount	no Priority	with Priority
1	101400,	105600
2	121900,	130200
3	137800,	166600
4	219900,	178400
5	205600,	198400
6	221900,	403200



4.5 Performance Metric and Proof

For our elevator, we wanted to measure total time it took to service requests. We measure elapsed time by some value t every time the elevator moves up of down one level. Our goal is to show that the elevator does not have to visit every floor for some building between 1 and n floors.

Proof. Claim there are floors m, l, k such that $1 < m < l < k < n$. We claim the elevator is on floor l, and the lowest requested floor is m, for some $m < n$, then the elevator does not have to visit floors 1 through $m-1$. Similarly, claim the highest requested floor is k for some $k < n$ and $k > m$. Then, the elevator also does not have to visit floors $k+1$ through n.

We claim SCAN will visit floors 1 through n at least once, and there will be some overlap of some length s. So, by adding our time unit t for each floor visited, we have

$$t * (n + s) = tn + ts \text{ units.}$$

For our algorithm, the elevator visits floors m through k at least once with some overlap of some length s'. As above,

$$t * ((k - m) + s') = (tk - tm) + ts' \text{ units.}$$

Since $(k - m) < n$, we know that $(tk - tm) < tn$, thus we conclude that our algorithm is faster than the basic SCAN algorithm. □

5 Conclusion

We acknowledge that this may not be the most optimized algorithm. There are many issues we had that were solved by adding more checks and loops. This algorithm also can take a good deal of real-time to complete depending on it's input size. Despite all this, the elevator algorithm will service all 'passengers' and follow priority order. In regards to improving this algorithm, we want to add in the functionality of this elevator being a real-time algorithm. This would significantly change our algorithm to be more life-like and much more useful. Also, we want to be able to add multiple levels of priority. Currently, floors either have priority or don't, but there could be cases where floor a has priority over floor b, and both have priority over all other floors. Finally, adding a limit to have many requests can be serviced at a time. This would translate into a weight or passenger limit in real life elevators.

In creating this algorithm, we thought hard about what was asked of us. Not only were we able to decipher and understand the typical SCAN algorithm, but also we created our own version of it that would be useful for the specific problem we wanted to solve. The fault with SCAN comes from two thing; SCAN moves from it's starting point to the end in both directions, most

likely hitting locations it doesn't need to check. Also, SCAN does not allow for requests to have different priority levels. These issues cause unnecessary waiting times and limit the amount of problems it can solve. We were able to work around these issues to create an algorithm that can get away with not moving all the way to the end of it's line. Our algorithm works when all requests have the same priority level and when certain floors have a priority over others. This two-way algorithm is great because it can be utilized in many settings.

References

[1] Bellis, Mary. "The History of Elevators from Top to Bottom." *ThoughtCo*, ThoughtCo, 11 Aug. 2019, https://www.thoughtco.com/history-of-the-elevator-1991600.

[2] Dunietz, Jesse. "The Hidden Science of Elevators." *Popular Mechanics*, Popular Mechanics, 3 Mar. 2021, https://www.popularmechanics.com/technology/infrastructure/a20986/the-hidden-science-of-elevators/.

[3] "Elevator Algorithm." *HandWiki*, 26 Nov. 2021, https://handwiki.org/wiki/Elevator%20algorithm.

[4] "History of Elevators." *Elevator History - Facts and Information*, 2022, http://www.elevatorhistory.net/.

[5] "Look Disk Scheduling." *Look Disk Scheduling*, 7 May 2020, https://www.tutorialandexample.com/look-disk-scheduling.

[6] Nihari, Nalli. "Look Disk Scheduling Algorithm." *GeeksforGeeks*, 6 Oct. 2021, https://www.geeksforgeeks.org/look-disk-scheduling-algorithm/.