Computer Science 303: Data Structures Project #1C

attachments and source available at https://github.com/alexskc/cs303

Aleksander Charatonik

October 8, 2018

1 Algorithm

The first step of designing an elevator simulator is to choose an appropriate algorithm, without worrying about implementation, which I will do here. I will start with the most straightforward algorithm that will solve the problem, then consider issues and possible optimizations.

The first and most basic solution is to simply take users to the floor they request, in the order the requests come in. The problem with this is immediately apparant. It's not very efficient. If the elevator is already heading up, and a user on the way is also heading up, it makes sense to pick them up along the way, rather than have them "wait their turn."

An alternative algorithm that would solve this problem would be to prioritize users going in the in the same up/down direction as the elevator. However, in a single-elevator setup, this is still sub-optimal for a user that is "almost on the way." For example:
1. Elevator is on floor 25
2. User on floor 1 requests elevator to a floor like 7.
3. Elevator starts going down to our user.
4. When the elevator is on floor 23, a user arrives on floor 24, requesting floor 1.

In this situation, continuing to get the first user would be sub-optimal. The 2nd user would have to wait for the elevator to move 63 floors before they get to where they want. By contrast, if we go back to get the 2nd user, the wait time for the first user only increases by 2.

We can fix this by prioritizing users based off how much they are expected to increase wait time, but there is a problem with this: we can get trapped in an infinite loop. For example, if, after the 2nd user showed up and got picked up, we kept adding a new user every 2 turns, the elevator would continue picking them up forever, without getting anyone where they intend to go.

In trying to find solutions to this problem, I was not able to find a solution by myself. After doing some research, I found that this is a similar problem to hard disk scheduling. Choosing which direction to send the elavator in is analogous to choosing which direction to send the disk head in. The algorithm I'm looking at is called "Shortest Seek Time First," and the problem of never getting to an old request is called "starvation." What I'm looking to develop is a "starvation-resistant SSTF." Of course, it's easy to design solutions to prevent starvation, such as prioritizing older requests, or some kind of hybrid algorithm. But thes e solutions just aren't very sound.

Moreover, SSTF doesn't always necessarily produce the shortest path. For example, if you have a lot of requests on both ends of the elevator, then the algorithm will be drawn to whichever is slightly closer, and then keep moving in that direction and take care of all the requests on one end before taking care of the other half. In these cases, it acts similarly to the standard elevator algorithm. The problem is this: It might've been more efficient to do the other half first. The algorithm has no way of knowing.

As we spend more time thinking about this, the problem becomes increasingly apparant: This is a variant of the Travelling Salesman Problem. We can make a few optimizations because an elevator is one-dimensional, but much of the problem still remains... Can we assume that our elevator has infinite computation power? Probably not. We can certainly use a heuristic to make the problem more manageable. With buildings as tall as the Burj Khalifa at a 163 floors, and 63 sectors per track on modern hard drives, we're certainly going to need one. We're certainly not doing 163! or even 63! problems. And one such heuristic is... The elevator algorithm! Back where we

started, let's write this one down formally:

1. If there are no pending requests, the elevator does nothing.
2. As soon as a request is made, the elevator moves toward the requestor. This decides its direction, ascending, or descending.
3. If there is a request in the same direction along the way, pick it up as well.
4. When there are no requests at the location of the elevator, or further along the direction of the elevator, going in the same direction as the elevator, switch directions.
5. Repeat until no requests are left. Stop the elevator until the next request, go back to 1.

Considerations

- Users are assumed to know the direction of the elevator. They only enter an elevator going in the same direction they are. In a single-elevator design, the wait time is the same regardless. In a multi-elevator design the wait can be shorter if a 2nd elevator arrives before the first elevator turns around.

- When an elevator arrives on a floor, all users waiting for it enter, not just one.

- Behavior can be very different depending on whether the amount of people that can be in an elevator at once is limited or unlimited. While the latter isn't realistic in real-world situations, ideally the software should be able to handle both cases.
- For multi-elevator designs, we have to consider which elevator will fullfill the request. This isn't anything more complicated than "the closest elevator that *can* fullfill it. (ie, heading in the same direction or standing still)"
- Multiple users should be able to show up and make requests simultaneously.

2 Implementation

We really only need to keep track of four things:

- The location of each user, either waiting on the floor, or in the elevator. If they get to their destination, we stop keeping track of that user.
- The desired location of each user. Desired direction can be inferred by (current location - desired location).
- The location of each elevator.
- The direction of each elevator.

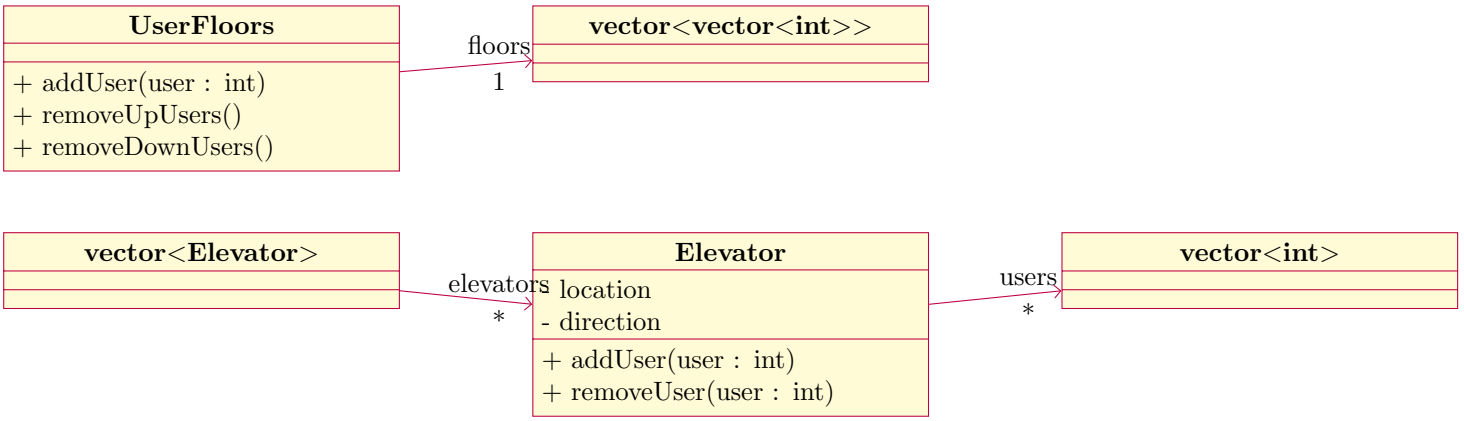
Multiple ways of storing these come to mind.

- Store an array of users, and give each user "current location" and "desired location," alongside an array of elevators, each with a "current location" and "direction."
- Store an array the size of the number of floors. Each cell represents a floor, and each floor contains an array of users. A second array stores elevators. Each elevator, in turn, stores a direction and an array of users. Users are a single integer, which represents their desired floor.
- A combination of the two. For example, using a floor array for users, but giving a location property to elevators.

I prefer the 3rd method. The 2nd is (subjectively) easiest to visualize, and also means there is no need to traverse the entire array of users if we want to figure out who is above/below the elevator. However, moving elevators becomes expensive if we have to keep copying them from one cell to another to move them. Below is a visualization, followed by a UML diagram:

	Users
20	
19	
18	
17	3
16	
15	
14	
13	
12	
11	15, 17, 20, 7
10	
9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

	Elevators
20	↓, NULL, NULL, NULL
19	
18	
17	
16	
15	
14	
13	
12	
11	
10	
9	↑, 20, 20
8	
7	
6	
5	
4	
3	NULL, NULL
2	
1	
0	



3 Interface

When designing the interface, we should keep several goals in mind:

- The user should be allowed to interact with the simulation interactively.
- It should be clear to the user what is happening at any given time
- That said, if the user wants to disable pretty-printing, they should also be allowed to.
- It should also be automatable and testable

The solution here is... Command line parameters! Going over our cases:

- If the user is running the program without any parameters, eg `./elevator`, then we give them the default interactive pretty-printed mode.
- If the user passes in `--no-pretty`, then the app skips printing out the elevator every turn.
- The user can pass in a parameter telling exactly what happpens in every turn, and skip all interactivity. The pretty printer will still show the state of the elevator every turn. This will look like `./elevator --sim="19 1:5:6:6 18-7:2-3:5-13 2-3 4-5:4-5"`. The first number indicates the number of floors. The next numbers,

seperated by colons, indicate the starting positions of each elevator. The number pairs afterwards `18-7` indicate users arriving at floors and requesting a floor. A space denotes the end of a turn.

- And finally, the user can pass in both `--sim` and `--no-pretty`, which will only print results such as longest/shortest waiting time, average wait time, etc.

The pretty-printing should look similar to the visualization shown above, albeit in ASCII.