Project 8

Added by Christopher Hoder, last edited by Christopher Hoder on Nov 13, 2011

Overview

The goal of this project was to implement the controls for a bank of elevators in a fancy hotel. We wanted to create a control software that picks up and delivers passengers to thier destination floor in the minimum average time per passenger. In this project we were supplied with most of the simulation code but needed to implement the Passenger and PassengerGroup classes inorder to get the basic simulation working.The goal was then to try and improve on the default elevator control algorithm. The PassengerGroup class works like a heapsorted priority queue. I used recursion to implement the private moveDown and moveUp methods. The code is very similar to the code we implemented in class for our integer heaps.

Solution

First I will discuss the implementation of the initial project requirements. My Passenger class follows closely to the documentation provided. My comparator classes: MaxFloor and MinFloor are implemented using generics so that there are less type casts in the code.

For the control strategies I created 5 different strategies. 2 of the strategies are strategies where each elevator acts individually and then 3 are strategies where the elevators are controlled by the elevator bank which takes into account all of the elevators collectively.All of the strategies are stored in an enum in the ElevatorBank class. They are static and public so can be accessed by all of the classes.

Individual strategies. These strategies are called INDIV and INDIV_2. The strategies are essentially the same except for one small difference when passengers are being let off at a floor. For the INDIV_2 strategy if all the passengers exit the elevator at a given floor, the elevator will check to see if any passengers are waiting to go in either direction before closing the doors. In INDIV the elevator only checks to see if there are passengers heading in the direction that the elevator was heading in.The rest of the strategy works as follows: The strategy is the same as the default case except for a few changes. In the nonEmptyRuyle the elevator will open the doors when a passenger is waiting to head in the direction the elevator is going at any given floor. The doors will not open however if the elevator is full. When an elevator is empty it will find the closest waiting passengers in either direction. If the passengers is not on the same floors as the elevator it will head toward the closest passengers.

Group strategies. For the group strategies, I started by adding the ElevatorBank to the landscape and making it extend SimObject (with no draw method). to simplify things I added a new SimObject data field in the Landscape to hold the bank. GROUP and GROUP_2 strategies are similar and GROUP_3 implements a fairly different strategy. All 3 algorithms start by determining which elevators are empty. Then for GROUP and GROUP_2 we will for each empty elevator assign it a targetfloor and direction to head once it gets to that target floor. The targetfloor will be determined by finding the Queue that has the most passengers waiting at it (out of any floor in any direction). Once an elevator has been assigned a targetFloor it will not be reassigned until either the passengers on that targetFloor are picked up by it or another elevator. Additionally, no 2 elevators will have the same targetFloor and targetDirection. The difference between GROUP and GROUP_2 strategies is that in GROUP_2 strategy the elevator will stop at any floor on the way to the targetFloor that has

Passengers waiting to travel in the direction of the elevator and when the targetDirection is also in the direction the elevator has to travel to get to the targetFloor. The GROUP strategy will not stop until it reaches the targetFloor. The group nonEmptyRule is similar to the individual strategies. GROUP_2 will also start with half of it's elevators at the top floor initially.

The GROUP_3 strategy works slightly differently. After determining which elevators are empty it will instead take the Queue with the largest number of waiting passengers and find the closest empty elevator to it. It will then assign that elevator's target floor and direction to that of the Queue's. Similarly to GROUP_2 strategy this will pick up passengers on the way if the same conditions are met. Addtionally, no two elevators will be assigned to the same queue as their targetFloors.

Also to change between strategies you need to either do it after the simulation starts using the key strokes or change it in the ElevatorBank class before compiling. It is set near the top of the constructor method.

Also to change between strategies you need to either do it after the simulation starts using the key strokes or change it in the ElevatorBank class before compiling. It is set near the top of the constructor method.

Below is the results for each simulation strategy with images.

Results:

The results of my 5 algorithms are as follows

INDIV: 200 out 200 in, total time 11450: [6 117 57.25]

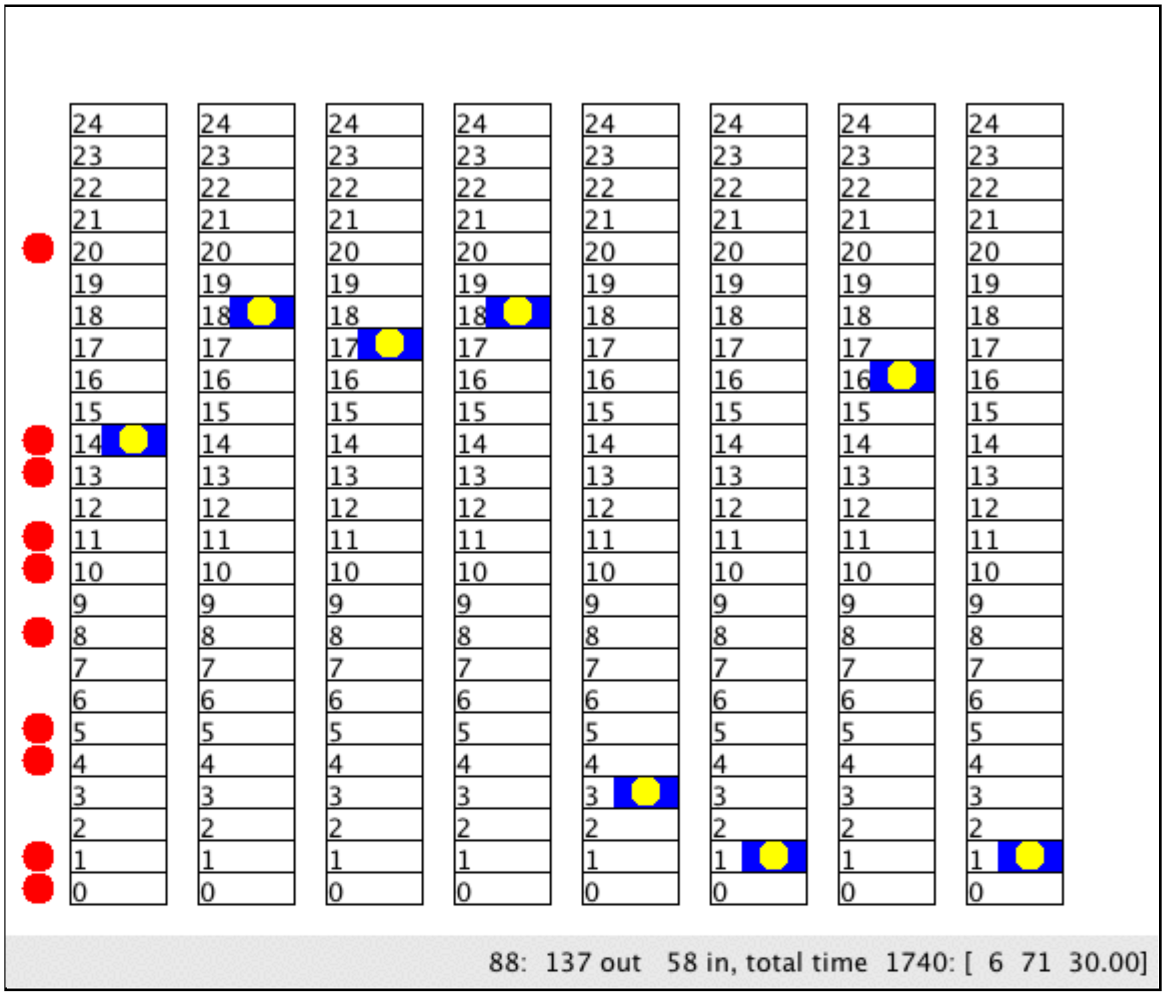
INDIV_2 200 out 200 in, total time 12054: [6 132 60.27]

GROUP 200 out 200 in, total time 10907: [6 121 54.54]

GROUP_2 200 out 200 in, total time 11120: [6 132 55.60]

GROUP_3 200 out 200 in, total time 11002: [8 155 55.01]

Below is a gif of my GROUP strategy working:



```
88:  137 out   58 in, total time  1740: [ 6  71  30.00]
```

Discussion:

So looking at the results of these 5 different control algorithms we can see that the group strategies all performed better than my individual strategies, which is to be expected because they all take into account information about the system as a whole and not just what they could know individually. We can easily see that some time is wasted by having multiple elevators travel to the same floor (especially in the beginning). The INDIV strategies also have some backtracking which adds to our overall time. We can see that using the Priority Queue's for the Queues helped us determine where to send the elevators such that we can minimize average time. I would have guessed that the GROUP_3 would be the fastest because it finds the elevators that are closest to the highest priority Queue. However, the increased time is probably due to sending Elevators that were probably close to high volume Queues far away to another Queue that is of only slightly higher size. Maybe with a different standard data set this strategy may be faster.

Extensions

For this project I implemented several different extensions. I created my own queue data structure to hold the Queue's on each floor. I did this so that I could then write a Priority Queue (QueueSorter) for these queues to find out which Queue had the longest line (or highest priority). After implementing my own ElevatorQueue class i realized that i could have just extended the linked list and added my extra data fields that way. The initial reason for creating my own class was to house the floor and direction (UP or DOWN) that each queue held passengers for. However the QueueSorter allows the elevator bank to decide which floor has the longest line. While this may not be the most feasible algorithm in real life it allows for slightly better average times than the individual averages (see results). We could imagine that there are weight sensors outside each elevator or something. This new algorithm was part of the extension where i created a few different strategies and compared them (See above results and discussion).

Additionally, I added some new keyboard controls to my simulation. I added the following buttons:

w: increases the simulation speed by 100ms

s: decreases the simulation speed by 100ms

r : will start saving every 10th iteration

e: will print out the current game statistics

d: this will move the simulation forward by 1 iteration (only if the game is paused)

1: sets the strategy to INDIV

2: sets the strategy to INDIV_2

3: sets the strategy to GROUP

4: sets the strategy to GROUP_2

5: sets the strategy to GROUP_3

Some of these might not be the most useful for using the simulation but helped with the developing of the algorithms and testing for errors.

In addition to these extensions I created documentation of all my code that can be seen in the provided html folder with my solution I submitted. As well as in the implementation of various methods made sure to use generics and exceptions where appropriate (see Elevator Queue and the passenger and PassengerGroup classes)

Conclusions

This project really helped me learn a lot about reading through commented code and learning how an already working algorithm actually works. I struggled for a long time to implement a ElevatorBank control algorithm using my QueueSorter. I found it required me to really understand the way the simulation was working initially as well as a good plan of action as to what I wanted to do in various different scenarios. I felt that coming up with an implementable control algorithm for this project was really hard because of the fact that you can't actually compute any optimal move at each iteration. We therefore need to decide upon some simpler strategy and implement that. I additionally used the provided code to learn some new parts of java including creating documentation and using keyboard controls. I felt that trying to implement these extensions helped me learn more about how the provided code works. I additionally learned how to implement priority queue's really well, doing it twice in this project. If I had more time I would have liked to combine them into one class using generics to specialize the two priority queues.

Labels

cs231f11project8