Cory Free

Professor Hare

CS 461

1 March 2020

Program 2 Report

My programs consists of 3 different classes and the source.cpp file. I have one class called schedule that essentially stores information for one class. It has a variable for the time, course name, the time, the professor, and the room. Then it also has getters and setters for each of these values. Then I have another class called fitnessFunction that is only used to calculate the fitness score for each full schedule. When an instance of this class is created, it requires a vector full of schedules to be included as a parameter. Then, in the constructor for the class, it calls all the fitness functions and sets the score automatically. I included more descriptions for the individual functions inside this class in the actual code. The final class I used is called testClass and it was only used to test the fitness functions. It does not currently produce any output, unless invoked, but it was helpful to make sure the functions were returning the correct score.

The first thing I thought to do was to create vectors with all the information for each item. I had vectors for all the professors, the classes that each professor could teach, the times, the rooms, the room capacities, and the class capacities. Then using the rand() function, I selected random items from these lists to create one full schedule. Each full schedule consisted of a vector full of instances of the schedule class. So each item in the vector would have its own course name, professor, time, and room. This made it pretty easy to iterate through the whole schedule to calculate the scores, change items, and whatever else I needed to do. Then, I had 3 iterators that were being incremented throughout the whole execution of the code. totalAttempts kept track of the total number of iterations regardless of whether or not a change was made. The attempts variable was used to keep track of how many iterations it had been since the last change. Every time a change was made it would be set to 0 and every time a change was not made it would be incremented by 1. Changes was incremented by 1 every time a change was successfully made and then it would be set to 0 whenever it was greater than 400.

Using the rand() function again, I then chose a random integer of 1, 2, or 3. This determined which category to change (1 for professor, 2 for time, and 3 for room). Then I picked a random integer again to be used to determine which value in the chosen category’s vector it would be swapped with. After creating a temporary object of schedule with the new change, it calculated the score of the new schedule with the old schedule. If the new one was greater, it would automatically make the change in the original schedule. If not then it would check the annealing function: randVal <= exp(-(original.fitnessScore - tempS.fitnessScore) / temperature). This function provides a way for the program to decide whether a schedule with a lower fitness score was worth travelling to. The higher the temperature, the more likely the program is to choose the new schedule.

The program would keep iterating through this loop, decreasing the temperature by multiplying by 0.9 every time the changes was greater than 400 or the totalAttempts was greater than 4000 to make the program less and less accepting of new schedules. It would stay in this loop until attempts was greater than 4000, or until there had been 4000 successful iterations without making a single change. This (hopefully) would return the schedule with the best fitness score. This, however, is where I ran in to some trouble. My program would never end no matter how long I let it run with these rules. I did a few things to get around this problem. For one, I made it exit the loop if the temperature reached a value of lower than 0.001. However, it would still take forever to run because it took forever for the temperature to reach this value. Another thing I tried was making it so that if the new score and the old score were the same, then it would not change it to the new schedule. After testing this a bunch, it produced basically the same fitness score in the end, and it only took around 8 minutes to run. The best score I could get was 286, and it is shown in the output.txt file included.

In conclusion, this was a very interesting project to work on, and it definitely helped me understand the concept of simulated annealing and search functions in general. One way that I could make my program better and faster would be to take out a few for loops. I used a lot of them in the fitness functions and I’m sure there are ways to replace at least a few of them with a more efficient search methods. One simple way that could be used to extend the program would be to simply add more classes, courses, times, etc. Another way would be to use another method of finding the best score besides simulated annealing. Then you could compare which method produces a better score on average, which one has better peaks, etc. Overall, I enjoyed this project and am feel as though it was a good way of putting the course material to use.