Stephen Townsend
CS 325-400
10/22/17

Assignment 4

Problem 1: Class Scheduling

Using the logic of greedy algorithm, I would start by sorting the start time of each of the classes in order from earliest to latest. From here I would look at the start times. We cannot have multiple classes in one lecture hall. As a result, there will be a certain depth to the program that will determine how many lecture halls will be used. This depth is determined by counting the number of classes whose duration time conflicts with other classes. Once we sort the set of classes the program and then start with the first class and assign it a number starting with one. This number is an indicator that a certain lecture hall is in use. This means that if one is currently used then the next class will have a different number. As the array is going from least to greatest and assigning a class to a lecture hall we can get the least amount of lecture halls used.

Algorithm

Sort the array of classes from least to greatest.

Use the first element and assign it a starting number.

When the class ends, the assigned number will be placed in the array for available halls.

Go to the next class, and use the next available hall.

Once there is an empty hall assign the number.

Time Complexity

Sorting the array of classes would be $\Theta(n\log n)$

Going through each class and checking the hall availability would give us $\Theta(2n)$. Given $\Theta(2n) < \Theta(n\log n)$ the running time would be $\Theta(n\log n)$.

Problem 2: Road Trip:

I would start by calculating a distance ratio to the hotels. Since we are looking for the shortest time to get to our destination we will have to drive as close to the max (d) miles. Using greedy algorithm on the list we would have to calculate the relationship percentage of total miles / hotel distance. I would then take the option that gets us closes to 1. As a result, take the lowest result of the total miles to distance ratio. After the ratio sort the numbers from lowest to greatest. The Closer the ration is to one the further we can travel. Example: Drive 900 miles per day. $h_1 = 200$, $h_2 = 150$, $h_3 = 500$. $900/200 = 4.5$, $900/160 = 6$, and $900/500 = 1.8$.

Time Complexity

Calculating the ratio $v_i = d_i/x_i$ would be $\Theta(n)$ because we need to check each element.

Doing the sort would take probably $\Theta(n\log n)$

The running time would end up being $\Theta(n\log n)$ because of the sorting needed.

Stephen Townsend
CS 325-400
10/22/17

Problem 3: Scheduling jobs with penalties:

The case is about having many jobs that need to be completed and each job has a penalty if it is not finished within the deadline. For this, to minimize the penalties, each job should be sorted in deciding order by the penalty amount where $p_1 \leq p_2 \leq \ldots \leq p_n$.

The best option to try and minimize the penalty given there is a chance that jobs can have the same d_i with different penalties. With the Greedy algorithm there is no way to check both d_i and p_i. As a result, the purpose is to try and get rid of the jobs that have the highest penalties first and disregarding the deadline.

The deadline will need to be checked against the current time count. If the count is > then the deadline then we will have to take the penalty and move on from that job to the next.

Algorithm

Sort the jobs in descending order based on the penalty.

Loop through the jobs and running them.

Once at the next job check the deadline. If the deadline is < time count then the job is skipped and the penalty occurs.

Time Complexity

The sort is going to be $\Theta(n\log n)$

Going through the list of jobs will be a single loop and that will be based on the elements in n. Thus, being $\Theta(n)$.

Problem 4: CLRS 16-1-2 Activity Selection Last-to-Start

The Last to Start is a greedy algorithm because greedy algorithms have:

> A set of candidate solutions, function that check if the candidates are feasible, selection function indicating at a given time which is the most promising candidate not yet used and, objective function giving the value of a solution.

With these conditions in mind. If we look at how Last to Start works, we have a set of data that can be part of the solution. In the problem being uses, the activities are the possible solutions. Each of the activities are checked based on certain conditions. If that condition is met then the best option is chosen at that point in time. From there the program moves forward checking the next activity. The program is only check the next one and is not making future predictions. Once the data is parsed then the solution can be outputted.

With all the above information, the Last to Start is greedy because it does not predict the future and chooses the best option at the current position of running. Once an option is selected then the program continues. The previous data is not changed at any point in time.

Stephen Townsend
CS 325-400
10/22/17

Optimal Solution.

Last to Start greedy algorithm yields an optimal solution if we are given a set S = (a1, a2, …, an) activities where $a_i = [s_i, f_i)$. We are looking for an optimal solution with the last activity that is compatible with the previous activities. If we instead create a set S'= (a'_1, a'_2, …, a'_n) activities where $a'_i = [f_i, s_i)$. If an $\in$ S and a'_n is the reverse would then be $\in$S'. Thus, an optimal solution for S is directly related to the reverse of S'

Problem 5: Activity Selection Last-to-Start Implementation

You may use any language you choose to implement the activity selection last-to-start algorithm described in problem 4. Include a verbal description of your algorithm, pseudocode and analysis of the theoretical running time. You do not need to collected experimental running times.

Pseudocode

```
Last-to-First(arr_line, arr_path, ((act_size*3)-1))
{
    currLocation = size-1;        //Current location to compare. End Time
    startTime = size - 4;         //Next Location to check. Start Time
    counter = 0;                  // Solution Counter

    arr_path = arr_line at currLocation – 2   //Path Starts at the last activity.  End Time - 2 = Activity #
    add one to counter

    For startTime <= 0 then decrease startTime          //Reverse through the array
        if currLocation -1 >= startTime
            Add (startTime – 2) to arr_path             //Activity Number
            Set currLocation to startTime
            Add one to counter
        Reduce StartTime by to 2                        //Move to next activity in list
    return counter                                      //Return # of activities scheduled
}
```

Algorithm

The Last to Start will be implemented by using a for loop. I believe I was making the problem harder by trying to figure out a way to sort the list. This would entail a 2D array I believe, but the implementation was not working. AS a result, I found that using a for loop to reverse through the list would give the greedy aspect of starting last and ending first. As a result, I took the last activity's (n) end time and compared it with n-1's start time. If the End time is greater or equal to the start time then I would save the current activity and move to the next activity that was found. If the activity's start is greater than the end time then my comparison position would advance up the list to the next activities start time.

Stephen Townsend
CS 325-400
10/22/17

Each activity that satisfied the above condition, the activity number would be placed in an array to hold the selection. Once the list is exhausted the implementation is finished. The selection array can then be outputted to the screen. I reversed through the array to get it is the order of start to finish.

Theoretical Running Time

The theoretical running time would be Θ(n) because I only needed to use a for loop to reverse through the array.