Lachlan Sinclair

7/21/2019

Homework 4

Problem 1:

I will be assuming the hotels are provided in order of increasing distance from the starting location. While not at the destination hotel, loop through the hotels with a greater distance than the hotel previously stayed at. While looping if the destination hotel is found (x_n), travel to it. Otherwise if a hotels distance from the previously stayed at hotel is greater than d (distance able to travel) select the previously looked at hotel.

Since no sorting is required the run time of this algorithm is Theta(n). Searching the nodes and backtracking immediately after exceeding the available distance means worse case each hotel is analyzed twice giving a total of 2n operations which is Theta(n) runtime. The best case is when the hotel can be reached in one day, however since we search the array in ascending order every element in the array will still be checked once making the worst case also Theta(n).

Problem 2:

First sort the jobs in decreasing order based on their penalty into array A, this will take nlogn time. Then create another array B of size n that will represent the order in which the jobs will be completed. Then starting with the first job j_i in A, try placing it in B at index d_i-1, if d_i-1 is unavailable look for any open position in B starting from index d_i-1 down to 0, if an open space exists place j_i into it. If an open space does not exist from d_i-1 to 0 in B, place j_i at the next open index in B starting from index n-1 moving back down to d_i. Repeat this process until every job in A has been processed. This will yield the optimal solution.

The runtime of my algorithm is $O(n^2)$. The upper bound to my algorithm is $O(n^2)$ and the lower bound is Omega(nlogn). Worse case occurs when an element is added to B and it must search over all previously added elements to find its place. Best case occurs when every j_i can be placed into B at d_i-1.

Problem 3:

By ordering the activities into an array by start times in descending order a greedy algorithm can be used to select activities. This is done adding the first activity in the array to the optimal set, then iterating through the rest of the array adding activities to the optimal set if their end time comes at or before the last activities start time in the optimal set. This algorithm tries to find an optimal solution on each iteration making it a greedy algorithm.

Proof: Based off Theorem 16.1 proof from the textbook. Proving this algorithm uses nearly identical logic, just switch earliest finish time with latest start time and the proof still hold.

Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the latest start time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the latest start time. If $a_j=a_m$ we are done, since we have shown that a_m is in some maximum size subset of mutually compatible activities of S_k. If a_j does not equal a_m, let the set $A'_k = A_k - \{a_j\}$ union $\{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the last activity in A_k to start, and $s_m >= s_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m.

The algorithm I described repeatedly chooses the activity that start last and is compatible with all previously selected activities until no activities remain. Which follows the above proof to select a_m values to be included in the maximum size subset, therefor it will correctly find a maximum size subset of activities.

Problem 4:

My code will read in the activities one set at a time. For each set it will create an array of structs, these structs will have int properties for the number, start time and end time of an activity. My code will then use a merge sort algorithm to sort the struct array in descending order of start values. It will then loop through the sorted array, assigning activities to the optimal solution when its end time is less than or equal to the start of the previous value in the solution.

Pseudo code:

```
int setNum=0

loop until the entire input file has been read
        int sizeOfSet <-read in set size from file
        setNum++
        struct activity arr[sizeOfSet]
        if(sizeOfSet <1) continue to next set
        For i=0 to sizeOfSet                                    (n)
                arr[i] <- read in the number, start time and end time to the i'th activities properties
        Merge sort arr by descending start value                (nlogn)
        struct activity solution[sizeOfSet]
        solution[0] = arr[0]
        int solutionCount =0
        for i=1 to sizeOfSet                                    (n)
                If(arr[i].end <= solution[solutionCount].start)
                        solutionCount++
                        solution[solutionCount] = arr[i]
```

```
display setNum
display solutionCount
for i=0 to solutionCount                                    (n)
        display solution[i]
```

Using this algorithm, the runtime of my code is Theta(nlogn), due to merge sort being Theta(nlogn). Further explanation: looking at the loops in my code the runtime is Theta(n)+Theta(nlogn)+Theta(n)+O(n) which gets dominated by the Theta(nlogn). Note: this is under the assumption that we are analyzing the algorithm runtime per set, if it was for x number of sets it would become, Theta(x*nlogn).