Programming Assignment Report 3

Dynamic Programming

(CSCI B505 Fall 19)

Manisha Suresh Kumar

UID:2000529881

10.17.2019

INTRODUCTION

Dynamic programming is an optimization technique that helps avoid revisiting sub problems as to obtain a faster run time. Dynamic Programming can have a top-down (memorization) or bottom up approach. Here, we are apply Dynamic Programming to the Job Selection problem.

SOLVING THE PROBLEM AND ANALYSIS

The problem requires the starting time of every new job to be greater than or equal to the previous job. Given this condition, we need to find a sequence of jobs that gives the maximum weight attainable. Therefore, the list of jobs was sorted first according to the finish times to give an order to the list and to make iteration through list shorter and less cumbersome. Sorting was done with merge sort ($O(nlogn)$).

Note:

- For input2.txt, since there are 100,000 jobs listed ,the merge sort takes approximately 47s to sort the list according to finish time.
- The input file name is to be passed as command line argument

Attempt 1

My first approach to solve the problem was using Top-down (memorization) and Dynamic Programming. For each job, the maximum attainable weight is found out by adding its weight to the maximum attainable weight that was found for the job after it. Dynamic Programming was implemented by storing these maximum attainable weights every time they are calculated so that whenever a job is confronted again they can be directly accessed. However, this required two for loops (n^2); one for finding the maximum attainable weight for each job, the other for iterating through the jobs to find the jobs that will satisfy the constraint. Additionally, there are approximately 'n' recursive calls too. This made for an asymptotic complexity of $O(n^3)$.

Attempt 2

Following this, I tried the bottom-up approach by propagating the maximum weight obtained so far from the start of the job sequence towards the end. Dynamic Programming was used to keep store of the maximum weights obtained at each step. When each job is

encountered, I would search the jobs up until the current job and that satisfies the constraint and add their weights to the current job. If no such job is found, the largest out of the current job's weight or the previous job's weight (which is the largest weight till that job) is taken as the new largest weight. At the end of the iteration, the last value in the dynamic programming structure will have the maximum weight attainable among the sequence of jobs. The asymptotic run time in this approach is accounted for by one for loop that iterates through the jobs and another for loop that iterates through the previous jobs to search for the first job that satisfies the constraint giving $O(n^2)$.

Though this method ran well for 'input1.txt', execution time for 'input2.txt', which has a large number of jobs, took too long and had to be terminated manually.

Attempt 3

Finally, in an attempt to reduce the execution time and complexity, after scouring online, I was introduced to the idea of replacing the linear search in the inner for loop with binary search on GeeksforGeeks[2]. This simple change greatly reduced the execution time and the new improved complexity obtained was (*O(nlogn)*).

CONCLUSION

Dynamic Programming helps avoid repetitively exploring sub problems and reduces run-time substantially. However, when the input size is large, we can still be faced with a long execution time, so a combination of dynamic programming and other optimization techniques such as the one for search will give a more efficient solution.

Our algorithm here runs at *O(nlogn)* (Complexity for both sort and finding job sequence)

And has a space complexity of *O(n)*

REFERENCES

1. Introduction to Algorithms by Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen

2. GeeksforGeeks: https://www.geeksforgeeks.org/weighted-job-scheduling-log-n-time/