Oregon State University

CS 325 - Analysis of Algorithms

Prof. Julianne Schutford

Spring 2021

Homework 3

Greedy Approach

by

Abdullah Saydemir

saydemia@oregonstate.edu

May 8, 2021

Q1)

a) The crucial part of the description is that the distances from the starting point are given in a *sorted* list. Given this list, a greedy algorithm would be as follows.

- Each day look at the map and highlight the farthest reachable hotel.
- Drive to that hotel, stay there at night.
- Do this until arriving at the destination.

This algorithm would minimize the number of days it takes to get to the destination. Also, we must assume that there is at least one hotel in reach every day.

Pseudocode for the algorithm is as follows.

```
FUNCTION roadtrip ( Vector x , int d )
    LET stayed_hotels <- new Vector()
    LET i <- 1
    LET position <- 0

    WHILE position != $x_n$ DO                // do until reaching to the end
        LET limit <- position + d             // max position I can travel to today

        IF $x_i$ > limit THEN                 // hotel cannot be reached
            RETURN [ ]
        END IF
        WHILE limit >= $x_i$ DO               // find the furthest hotel within reach
            position <- $x_i$
            i++
        END WHILE
        stayed_hotels.insert( i )             // mark it to stay at night
    END WHILE
    RETURN hotels
END FUNCTION
```

I would like to explain the pseudo code since some parts may not be clear.

```
IF $x_i$ > limit THEN                    // hotel cannot be reached
    RETURN [ ]
END IF
```

This if check is necessary. If the gap between two hotels is greater than the distance we can travel per day, than we cannot complete the trip. For example,

x = [50, 100, 175, 200, 250]
d = 50

First night, we will stay in the first hotel and second night in the second hotel. However, on the third day we will not reach the next hotel since the *$d < 175 - 100$.* Therefore, the function returns an empty list rather than the hotels we stayed in.

b) Running time of the algorithm is $\Theta(n)$.

There are two important points in the discussion. First, the input list is sorted. If it was not sorted, then we had to sort the list and that would make the running time $\Theta(n\, logn)$.

The second point is that we assume there is at least one hotel in reach every day. If this is not the case, the algorithm will return quicker for some inputs. In that case, we cannot have this algorithm's running time in Θ notation. Instead, we have to use O notation.

Nested while loops may trick us to conclude that the running time is $\Theta(n^2)$. However, these while loops iterate over the same vector and the end condition is reaching to the end of this vector. Therefore, the running time is equivalent to traversing the list one time, which is $\Theta(n)$.

I don't know why the full marks is given to the algorithm with $\Theta(n\,logn)$ running time in the rubric. There was a similar question in the class and the difference is that that question had a penalty mechanism and we were to minimize the penalty not the number of days. In that case the running time was $\Theta(n\,logn)$. However, the question in the homework does not have such a mechanism. So, I believe that something is confused with that problem.

Q2)

A greedy solution maximizes the utility in each step and never looks back. Last to start is a greedy approach because it maximizes the "available time" interval for other activities by choosing an activity. Informally, it tries to justify the activities to the right so that it can fit more activities into the available time interval on the left side.

The algorithm has the optimal substructure property. When you sort the activities based on the start time, last to start will always choose the last activity in the list because we know that it has the latest starting time.

Definitions

- a_i : Activity *i*
- s_i : Start of the activity *i*
- f_i : Finish of the activity *i*
- S_{ij} : Set of activities that can occur between the completion of a_i and the start of a_j
- A_{ij} : The maximal set of activities for S_{ij}
- $c[\,i,j\,]$: Size of an optimal solution for the set S_{ij}

Problem Statement

Suppose we have a set $S = \{ a_1, a_2, \ldots, a_n \}$ of *n* proposed activities that wish to use a resource which can serve only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i ,where $s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i , f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i , f_i)$ and $[s_j , f_j)$ do not overlap. In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of start time:

$s_1 \leq s_2 \leq s_3 \leq \dots \leq s_{n-1} \leq s_n$

Optimal Substructure Property of Activity Selection

Let us denote $S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$ and suppose that we wish to find a maximum set of mutually compatible activities in S_{ij} , and suppose further that such a maximum set is A_{ij} , which includes some activity a_k. Then, using a "cut-and-paste" argument, if A_{ij} contains activity a_k then we can write

$A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$

where A_{ik} and A_{kj} must also be optimal. *(Otherwise, it would contradict the assumption that A_{ij} was optimal)*

With this way of characterizing optimal substructure, if we denote the size of an optimal solution for the set S_{ij} by *c[i,j]* then the recursive solution to compute it would be like this:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \, , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \, . \end{cases}$$

Greedy Choice Property of Activity Selection

If we make the greedy choice, we have only one remaining subproblem to solve: finding activities that finish before a_n starts. Why don't we have to consider activities that start after a_n finishes? We have that $s_n < f_n$, and s_n is the latest start time of any activity, and therefore no activity can have a start time greater than or equal to f_n . Thus, all activities that are compatible with activity a_n must finish before a_n starts.

Furthermore, we already said that activity selection already exhibits optimal substructure. Let $S_k = \{a_i \in S_k\colon f_i \leq s_k \}$ be the set of activities that finish before activity a_k starts. If we make the greedy choice of a_n then S_n remains the only problem to solve.. Optimal substructure tells us that if a_n is in the optimal solution, then an optimal solution to the original problem consists of activity a_n and all the activities in an optimal solution to the subproblem S_n.

Proof that produced solution is correct

Theorem

Consider any non-empty subproblem S_k with activity a_m having the latest starting time. Then, a_m included in some maximum-size subset of mutually compatible activities of S_k.

Proof

Let A_k be an optimal solution for S_k and a_j be the activity in A_k with the latest start time.

- If $a_j = a_m$ then the condition holds.
- If $a_j \neq a_m$ then construct $A_k' = A_k - \{ a_j \} \cup \{ a_m \}$.

Since $s_m \geq s_j \Rightarrow A_k'$ is still optimal.

The activities in A_k' are disjoint since the activities in A_k are disjoint, a_j is the last activity in A_k to start and $s_m \leq s_j$. Since $|A_k| = |A_k'|$, we conclude that A_k is a maximum-size subset of mutually compatible activities for S_k and include a_m.[1]

Another proof would be having first to finish proof and reversing the time, but mathematics does not work that way :)

[1] I used the wording, the notation and a picture of an equation from the course book *Introduction to Algorithms (CLRF, Third Edition)* and the course slides. I converted some sentences and notations to prove that last-to-start is also a greedy solution to the activity selection problem.

Q3)

Description:

Given id, start time and finish time of activities, for each test case, last to start activity selection can be described as follows.

- Store all activities in an Activity struct that has id, start time and finish time attributes.
- Put all activities in a vector.
- Sort them according to the start time in ascending order.
- Take the last Activity in the vector since it has the latest start time.
- Go back in the vector and find other Activities that does not collide with the last selected Activity and have the latest start time.
- Print the results.
- Do this until reaching the end of the file.

Pseudocode:

I used the same merge sort algorithm from the first homework assignment. It sorts the Activities according to the starting time. So, I will not include the pseudocode for that.

```
STRUCT Activity
    INT id, start, finish
END STRUCT

FUNCTION last_to_start ( Vector sorted_activities )
    LET n <- length ( sorted_activities )
    LET selected_activities <- new Vector()
    selected_activities.insert( last element of sorted_activities )  // last element is always
                                                                     //  included
    LET itr <- n-1
    FOR m <- n-2 to 0 DO                                              // start from the end
        IF sorted_activities[ m ].finish <= sorted_activities [ itr ].start THEN
            selected_activities.insert ( sorted_activities[ m ] )       // take if the activity
            itr <- m;                                                   // does not collide
        END IF
    END FOR
    RETURN selected_activities
END FUNCTION

FUNCTION main()
    FILE input := read(act.txt)
    IF file is NOT correctly opened THEN
        print "Error reading file"
        exit
    END IF

    LET test_case <- 1                                    // Set number
    LET test_size <- 0                                    // number of activities in this set
    WHILE test_size <- cin DO                                   // read number of activities
        LET all_activities = new Vector<Activity>()

        FOR i <- 0 to test_size DO                                  // read the attributes
            LET id <- 0
            LET start <- 0
            LET finish <- 0
            all_activities.insert ( new Activity<id, start finish> )     // put them to a vector
        END FOR

        // sort activities based on start time in ascending order
        LET sorted_activities <- merge_sort( all_activities )
        // run last to start greedy algorithm and take the result
        LET selected_activities <- last_to_start ( sorted_activities )

        // Print the results
        LET size <- length ( selected_activities )
        print "Set " + test_case
        print "Maximum number of activities = " + size
        FOR i <- size to 0 DO
            print " " + selected_activities[ i ]
        END FOR
        Clear all the variables
        test_case++
    END WHILE
END FUNCTION
```

Explanation

In main, id, start time and end time of the activities are read from the act.txt file and saved into a vector. Then, activities are sorted based on the start time by using merge sort in ascending order and sorted list is given to the last-to-start function.

Explanation is the same as in **Q2**. Algorithm right justifies the activities based on the start time by choosing the last-to-start activity that does not collide with any other activity chosen before. By doing that, it opens up space on the left hand side so that it can put more activities.

Theoretical Running Time

Last-to-start function iterates over the activities one time and spends constant time on each activity. So, the running time of the algorithm is $\Theta(n)$ if we are given a list of activities sorted according to the start time. In any other case, we have to sort the list. Merge sort takes $\Theta(n\, logn)$ time and makes the overall running time $\Theta(n\, logn)$ as well.