CMSC 37000: Homework 2

Caleb Derrickson

February 9, 2024

Contents

Problem 1

We are given a set of intervals $I_1, ..., I_n$ on the real line. Each interval I_i is specified by the start and end points s_i and t_i, respectively:

$$I_i = [s_i, t_i]$$

Additionally, for each interval I_i, we are given a color c_i which is either red or black. We want to find a subset of intervals A so that

- no two intervals in A intersect;
- there is an equal number of red and black intervals in A;
- the total length of all intervals in A is maximized.

Design a polynomial-time dynamic programming algorithm that finds an optimal solution. Describe your algorithm in detail. Prove its correctness. Specifically, do the following:

1. Define a dynamic-programming table and explain the meaning of its entries.
2. Write the initialization step of your algorithm.
3. Write the recurrence formula for computing the entries of the table.
4. Explain the formula.
5. Find the running time of your algorithm.

Solution:

First, to ensure we are working with reasonably ordered subintervals, we need to sort all intervals according to their starting time. Assume that we have n intervals, of various lengths. The DP table, denoted as DP, will then be designated as a 2D table. DP will have rows up to the number of subintervals, n, where row i will consider the first i rows in the interval. Since there are only two colors considered, we only need to be concerned with storing the number of, say, red intervals. DP's columns will then carry the total number of red intervals used.

We will next explain the initialization of our DP table. Naturally, if we have not considered any intervals yet, we should have a maximum possible length of zero. We therefore set DP[0][j] = 0 for all j. Next, we need to consider the total number of black subintervals which can be chosen after each consideration i. So for every row i, the j-th elements need to be chosen carefully: if j is greater than the remaining number of black subintervals, we will initialize this as a -1, since this would be an unfeasible pairing. Furthermore, by the restriction that we require the same number of red intervals as black intervals, we set all elements in the 0-th column to -1, where we exclude the [0, 0] case (it is already set to zero).

I will explain my rationality in my recurrence formula while building it up / considering all cases. Suppose there is some optimal choice of subintervals S'. When considering interval i, we then have two choices:

1. $i \notin S'$.
2. $i \in S'$.

Considering the first case, where we exclude i from S', the best selection of subintervals has already been chosen up to and including subinterval i. Thus DP[i][j] = DP[i-1][j]. The second case, where $i \in S'$, is then broken down based on the coloring of interval i. If interval i is red, we first need to consider if there is another possible black interval we can choose, to maintain the equal number of red and black intervals. If this is satisfied, and there is a viable black interval we can choose in some possible future iteration, we update the DP table as follows: DP[i][j] = $\max\{$DP[i][j], DP[i-1][j-1] + $(t_i - s_i)\}$. If the interval i is instead black, we need not worry about violating the equal number of red and black intervals, since we are already considering such a specific number of red intervals already in the recursion. Thus, the DP table will be updated in this manner: DP[i][j] = $\max\{$DP[i][j], DP[i-1][j] + $(t_i - s_i)\}$. Thus, the recursion formula is

$$\text{DP}[i][j] = \begin{cases} \max\{\text{DP}[i][j], \text{DP}[i-1][j-1] + (t_i - s_i)\}, & \text{Interval } i \text{ is red} \\ \max\{\text{DP}[i][j], \text{DP}[i-1][j] + (t_i - s_i)\}, & \text{Interval } i \text{ is black.} \end{cases}$$

Therefore, when choosing the collection of subintervals which maximizes the length and maintains balance between red and black intervals, we consider the last row of the DP table. Each entry of the last row considers a different number of allowed red intervals, thus taking the max over the last row will return the best possible total length of subintervals. The only two places which will bottleneck us in terms of running time is the sorting of the intervals, and filling out the DP table. The sorting of subintervals can be done in, at best, O($n \log\ n$) time, whereas the DP table needs a running time of O(n^2) to compute all entries. Therefore, the running time is O(n^2).

Problem 2

Consider a rooted binary tree T, in which every vertex u is labelled with an integer number x_u. We say that T satisfies Property **H** if

$$\text{for every vertex } u \text{ and every child } v \text{ of } u,\ x_u \geq x_v.$$

Consider the following problem: given a tree T, change the smallest possible number of vertex labels x_u so that the tree satisfies property **H**. When we change the label of u, we can assign x_u any integer number we want. Design a DP-algorithm that solves this problem. Specifically, do the following:

1. Define subproblems.
2. Define the dynamic-programming table and explain the meaning of its entries.
3. Write the initialization step of your algorithm.
4. Write the recurrence formula for computing the entries of the table.
5. Explain why the recurrence formula is true.
6. Find the running time of your algorithm.

Solution:

1. Defining subproblems:

   Starting from the root of the tree, we can consider the two subtrees, labelled T_L and T_R, which have its children as roots. The analysis preformed on the tree T is then equivalent to the analysis preformed separately on T_L and T_R, bar the case where we consider the root of T with the roots of T_L and T_R. We can then repeat this delegation of subproblems up to the leaves of the tree T. This will then be treated as a separate case in recursion.

2. The DP Table:

   In order to keep track of the two subtrees T_L and T_R, we need to establish two tables, labelled $L[n]$ and $R[n]$ respectively, in which the entries of the tables respect the decision made at each iteration of the algorithm. We can also define another DP table, call it $A[n]$, which will have entries of our decision made at each iteration. Note that the table $A[n]$ is necessary, as the updating of $L[n]$ and $R[n]$ will be handled separately.

3. Initilization:

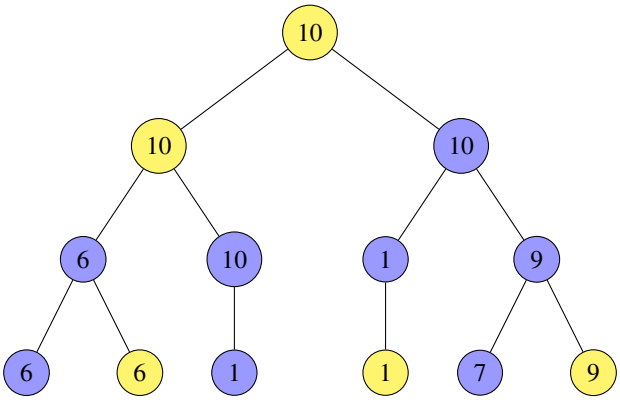
   The initialization of our algorithm basically translates to our handling of the leaves of T. Since we wish to establish a hierarchy on the tree, meaning the values of the nodes at depth d bound the values of the nodes at depth $d+1$. Note that, in the case of leaves, we are only considering the value of the leaf x_v and its parent x_u. To this extent, we will take the minimum of these two values, and set the new node value to it. That is, in the case where u is a leaf of T, with parent v, $x'_u = \min\{x_u, x_v\}$.

4. The Recurrence Formula:

   Since we are considering the case where u is a leaf separately, we can assume that the vertex u has children v_L and v_R. Our three DP tables make the handling of this case easy, since we can take $x'_u = \max\{x_{u_L}, x_{u_R}\}$. Note that the values of x_{u_L} and x_{u_R} are the updated values from a previous step in recursion. The DP tables are then updated as follows: supposing vertex u is the root node considered at iteration i, then $A[i] = \max\{L[i-1], R[i-1], T[i]\}$, where $T[i] = x_u$. The update of L will be $\max\{L[i-1], T[i]\}$, with a similar update schema for R.

5. The Algorithm's Validity:

   By the structure of the proposed algorithm, we have established nodes of greater depth will have smaller values than nodes at lower depths. Thus, our algorithm will produce a tree which comports with the desired result. The minimum number of value modifications is also guaranteed, due to our separate handling of the leaves of T. Note that, if we take the example tree as a test for our proposed algorithm, the nodes modified are different than the proposed solution. Nevertheless, the number of changed node values is similar. If we run the proposed algorithm on the given example, we will get the result given below. Note that the number of changed node values, 5, is equal to the given solution. I have colored the nodes in a similar manner. Note that my algorithm returns the accessed values in DFS, the number of changed nodes can just be obtained when comparing the values of the array A and the original tree T.



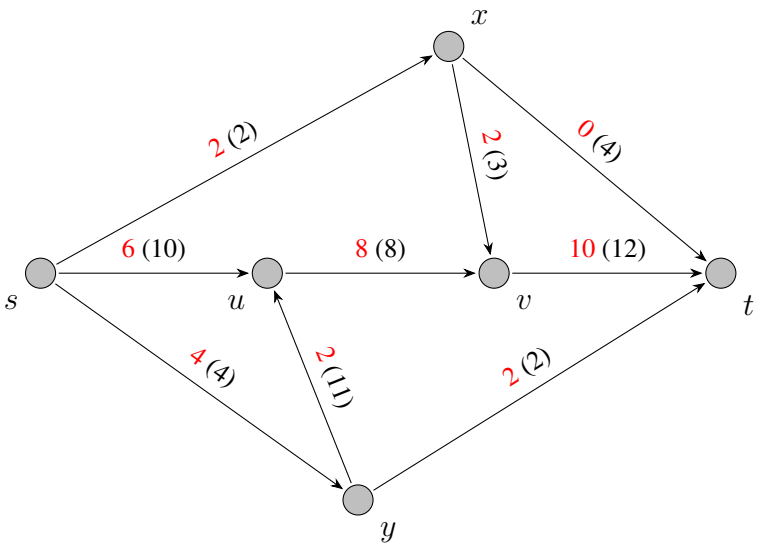
6. The Running Time:

   By the structure of our algorithm, we will need to traverse each node a linear number of times. The same can be said for the edges. Thus our running time is $O(|V| + |E|)$, where V is the set of vertices in T and E is the set of edges of our tree. Note that looping over the arrays A and T is done in linear time, scaling on the number of vertices. This bound will be less than the calculation of the DP tables, thus discarded.

Problem 3

Consider the flow network G and flow f shown in the figure below. For every edge e, the slow amount $f(e)$ on the edge and its capacity are written next to e as $f(e)$ $(c(e))$.

- Draw the residual graph G_f (draw all forward and backward edges of G_f). For every edge, indicate whether it is a forward or backward edge.
- Write the residual capacity of every edge e of G_f (you can write it next to the drawing of the edge).



Solution:

I will include the residual graph below, which will have its backward edges dashed, and the forward edges solid (this is the same convention as shown in class). To create the residual graph G_f, we consider each edge separately in G. If the edge is fully saturated, we replace the original edge with a backward edge of the same flow value. If the edge is not saturated, i.e., the flow along that edge is less than its capacity, we replace the existing edge with a forward edge; with flow value $f_e - c(e)$; and a backward edge; with flow value f_e.