Homework 10

Brian Knotten, Brett Schreiber, Brian Falkenstein

September 21, 2018

8

Assume the vertices are ordered via a breadth first search or a depth first search. This ensures a list of vertices from v_1 to v_n such that any vertex v_i is connected to one and only one vertex in $v_1 ... v_{i-1}$. v_i cannot be connected to more than one vertex in $v_1 ... v_{i-1}$, otherwise there would be a cycle and T would not be a tree. v_1 must be connect to a vertex in $v_1 ... v_{i-1}$ by definition of a tree search. Let $neighbor(v_i)$ represent the sole neighbor of v_i of $v_1 ... v_i$. Let w_i be the weight

# Base case: When T has one vertex, the most profitable (and only) thing to do is stay put. Return 0 for the best profit.
MostProfitableEndpoints[1] = $(1, 1)$
MaximumProfit[1] = 0

MostProfitablePathRootedAt[1] = v_1 # The path is (1, 1): the start and end point
RootedProfit[1] = 0

for i := 2 to n do:
    if RootedProfit[neighbor(v_i)] $+w_i > 0$ then:
        RootedProfit[i] = RootedProfit[neighbor(v_i)]
        MostProfitablePathRootedAt[i] = MostProfitablePathRootedAt[neighbor(v_i)]
    else: # Stay put
        RootedProfit[i] = 0
        MostProfitablePathRootedAt[i] = v_i

Let u, v = MostProfitableEndpoints[i - 1]
    if neighbor(v_i) $\in$ (u, v) and $w_i > 0$ then: # v_i adds value to the optimal solution
        if neighbor(v_i) == u then: MostProfitableEndpoints[i] = (v_i, v)
        else then neighbor(v_i) == v: MostProfitableEndpoints[i] = (u, v_i)
    MaximumProfit[i] = MaximumProfit[i - 1] + w_i

      else if RootedProfit[i] ¿ MaximumProfit[i - 1]: # v_i could possibly still be in the best path.
        MaximumProfit[i] = RootedProfit[i]
        MostProfitablePathRootedAt[i] = MostProfitablePathRootedAt[i - 1]

    else: # v_i doesn't contribute at all to the maximum profit path
        MaximumProfit[i] = MaximumProfit[i - 1]
        MostProfitableEndpoints[i] = MostProfitableEndpoints[i - 1]

Output MostProfitableEndpoints[n]

9

An algorithm for this problem is a modified form of the Optimal Binary Search Tree dynamic algorithm. Like the OBST dynamic programming algorithm, this algorithm caches the minimum weight of a tree with keys K_l and K_r in a 2D array, Weight[l][r]. It also stores the key K_i where $l \leq i \leq r$ that should be used as the root of the OBST for keys from K_l to K_r in a 2D array, Root[l][r]. Our algorithm differs from the OBST algorithm in that there is a third 2D array, Height[l][r] which stores the height of the OBST with keys from K_l to K_r.

The base case is the same as with the OBST algorithm. When $l = r$, that is, on the diagonal of the 2D array, the only key that can be considered as the (childless) root is K_l. So Root[l][r] = K_l and Weight[l][r] = w_l. Moreover, since the sole key does not have any children, Height[l][r] = 1.
The recursive case considers all keys from K_l to K_r. Like the OBST algorithm, consider the key K_i that has the minimum weight of the sum of its subtrees Weight[l][i-1] + Weight[i + 1][r] plus the sum of the weight of all the keys K_l through K_r, and make Root[l][r] = K_i. But do not consider K_i if Height[l][i - 1] - Height[i + 1][r] ¿ 1 or ¡ -1. Discard K_i as a candidate root if this is the case.
As with OBST, output the tree from K_0 to K_n. This algorithm is very similar, only it requires caching a third property of the subtrees, and has greater scrutiny over candidate roots.