University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
S. Faour, M. Fuchs, A. Malyusz

Algorithm Theory
Sample Solution Exercise Sheet 4

Due: Friday, 17th of November, 2023, 10:00 am

Remark: You are required to use the principle of dynamic programming in all of your algorithms. It is preferable if you write the algorithms in pseudocode and to write down the recursions.

Exercise 1: Leaf to Leaf *(10 Points)*

Given a rooted tree $T = (V, E)$ with n nodes such that each node is assigned a *unique* integer value. Note that we define a *rooted tree* to be a tree where one node has been designated the root.

(a) Find an algorithm that runs in $\mathcal{O}(n)$ and computes the maximum sum of the node values from the *root* to any of the leaves without re-visiting any node. Argue correctness and analyze its running time.

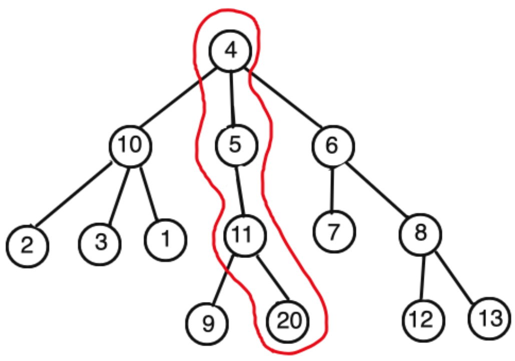

Figure 1: This is an example to our problem where the node with value 4 is the root. Thus, the path marked in red has the maximum sum of values of nodes in its path from the root to a leaf, hence your algorithm should output 4+5+11+20=40.

(4 Points)

(b) We generalize the problem and ask you to design an algorithm that runs in $\mathcal{O}(n)$ and calculates the maximum sum of the node values from *any leaf* to any other leaf without re-visiting any node. Argue correctness and analyze its running time. *(6 Points)*

Sample Solution

First consider $T = (V, E)$ to be a rooted tree with node $r \in V$ as its root. For convenience, we define $T(r)$ to be the subtree in T rooted at $r \in V$. To have a parent and child reference for every node, we do a preprocessing step where every node will have a pointer reference to its parent and that can be done in $|E| = n - 1 \in \mathcal{O}(n)$ time (e.g. one may start from the designated root and by doing a BFS one can orient every edge towards the root). Let $value(r)$ denote the unique designated value of node r. Let $N_c(r) = \{v \in V \mid v \text{ is a child of } r \text{ in } T\}$.

(a) **Algorithm (and correctness) idea:** Let $OPT(r)$ denote the maximum sum of the node values from the *root* r to any of the leaves in $T(r)$ without re-visiting any node. Assume w.l.o.g that $P_{r,v}$ is the path starting from root r, ending with leaf v in $T(r)$,and the (unique) optimal path for our problem i.e. the sum of the nodes values on $P_{r,v}$ is $OPT(r)$ (it is unique because all integers are assumed distinct). Therefore, the portion of $P_{r,v}$ starting from the child of r which is on $P_{r,v}$ to leaf v must itself be the *best* optimal path from any child of r to any leaf in $T(r)$, otherwise we can construct another path from the root r to a leaf in $T(r)$ whose sum of its path nodes values are larger than that of $P_{r,v}$, thus $P_{r,v}$ is not optimal, which is a contradiction. Hence, we can solve the problem via the following recursion:

$$OPT(r) := value(r) + \max_{u \in N_c(r)} OPT(u), \;\; OPT(r) = value(r) \texttt{ if } V := \{r\}$$

We propose the following algorithm using memoization:

Algorithm 1 `Root2Leaf(`r`)` ▷ assume $\texttt{memo}_1$ is a global dictionary initialized with `Null` and given $T(r)$

```
    if $T(r)$ is only node $r$ then return $value(r)$                          ▷ base case
    if $\texttt{memo}_1[r] \neq$ Null then return $\texttt{memo}_1[r]$                 ▷ decision was made before
    $\texttt{memo}_1[r] \leftarrow value(r) + \max_{u \in N_c(r)}$ Root2Leaf(u)                 ▷ Memoization
    return $\texttt{memo}_1[r]$
```

Running time: A naive running time analysis would be the following: we have n subproblems all in all (i.e. one for each node where they are of the form $OPT(r)$) and each is computed once (in general due to memoization). Moreover, for each subproblem we have to do one addition and find the maximum value of at most Δ values i.e. finding the maximum amongst each node's children, where Δ is the maximum degree of a node in the given graph thus giving us a $O(n\Delta) \in O(n^2)$ running time.

A linear time running time analysis can be achieved by noticing that each node v has its $\texttt{memo}_1\texttt{[v]}$ or $OPT(v)$ value read by only its parent at most once. Moreover, the cost of the whole algorithm can be seen as the sum of all these reading steps (up to a constant multiplicative factor since we have e.g. some constant arithmatic operartions to do per each node as well) and we have n of those, thus we have a running time of $O(n)$[1].

NB One may notice that storing the results of the subproblems is not necessary i.e. memoization and thus dynamic programming approach don't seem necessary, since in the whole algorithm we only call for each subproblem one time, thus a recursive algorithm is enough to solve the problem in $\mathcal{O}(n)$. However, asking for a dynamic programming algorithm for this part is appropriate and helpful if we want to solve the upcoming part.

(b) **Algorithm (and correctness) idea:** Let $OPT'(r)$ denote the value of the maximum sum of the node values from any *leaf* u to any other leaf v in $T(r)$ without re-visiting any node in $T(r)$. Let $T(r_{\text{rest}})$ denote the subtree $T(r)$ after removing all nodes (except root r) that belong to the root to leaf optimal path in $T(r)$ (note that this process can created disconnectthe graph, so we also remove any node that gets disconnected from the root r afterwards). Let $OPT(r_{\text{rest}})$ denote the value of the maximum sum of the node values from root r to any other leaf v without re-visiting any node in $T(r_{\text{rest}})$.

[1]It may also help to think of how a bottom top approach solution works i.e. when we start memoizing from the leaves and add the maximum of leaves to the root of every sub-tree. At the last step, there will be the root and the sub-trees for each child under it, adding the value at the node and the maximum of sub-tree will give us the maximum sum of the node values from root to any of the leaves.

Now, we observe the following: either r is on the optimal path of our leaf to leaf problem in $T(r)$ or it is not. Hence, we can solve the problem via the following recursion:

$$OPT'(r) := \max \begin{cases} OPT(r) + OPT(r_{\text{rest}}) - value(r) & \texttt{, if } r \texttt{ is involved} \\ \max_{u \in N_c(r)} OPT'(u) & \texttt{, if } r \texttt{ is not involved} \end{cases}$$

and for the base case $OPT'(r) = value(r)$ `if` $V := \{r\}$

Note that we can easily modify Algorithm 1 on input $T(r)$ to get a new $O(n)$ dynamic programming algorithm that gives us the value of $OPT(r_{\text{rest}})$ and we call this function `Root2Leaf(`$r_{\texttt{rest}}$`)` (the idea is to first run Algorithm 1 on input $T(r)$ and keep track of the nodes on the optimal path then remove them, afterwards we can run again Algorithm 1 on the remaining subtree rooted at r).

Algorithm 2 `Leaf2Leaf(`r`)` ▷ assume $\texttt{memo}_2$ is a global dictionary initialized with `Null` and given $T(r)$

```
if $T(r)$ is only node $r$ then return $value(r)$                          ▷ base case
if $\texttt{memo}_2[r] \neq$ Null then return $\texttt{memo}_2[r]$          ▷ decision was made before
$\texttt{memo}_2[r] \leftarrow \max \begin{cases} \texttt{Root2Leaf}(r) + \texttt{Root2Leaf}(r_{\texttt{rest}}) - value(r) \\ \max_{u \in N_c(r)} \texttt{Leaf2Leaf}(u) \end{cases}$   ▷ Memoization
return $\texttt{memo}_2[r]$
```

Running time : A similar running time analysis as above will give us linear in n running. We have here $3n$ subproblems all in all (i.e. 3 for each node where they are of the form $OPT(r), OPT(r_{\text{rest}}), OPT'(r)$) and each is computed once (*due to memoization*). Moreover, each node v has its `memo[v]` values for `Root2Leaf(`v`)`,`Root2Leaf(`$v_{\texttt{rest}}$`)`,`Leaf2Leaf(`v`)` used by only its parent O(1) number of times. Finally, the cost of the whole algorithm can be seen as the sum of all these reading steps (up to a constant multiplicative factor since e.g. we also have some constant arithmatic operartions to do per each node as well) and since we have $3n$ of these reading steps, thus we have a running time of $O(n)$. [2]

Exercise 2: Mutually Involved Bitstrings *(10 Points)*

Consider the following bitstrings $A = a_1a_2...a_m$, $B = b_1b_2....b_n$, and $C = c_1c_2...c_{m+n}$. We say that A and B are *mutually involved* in C if C can be obtained by rearranging the bits in A and B in a way that maintains the left-to-right order of the bits in A and B.
For example $A = 010$ and $B = 10$ are mutually involved in $C = 01100$ but not in 00101.

Give an algorithm that runs in $\mathcal{O}(m \cdot n)$ and determines whether A and B are mutually involved in C or not. Argue correctness of your algorithm and analyze its running time.

Sample Solution

Algorithm (and correctness) idea: For convineance, we can redefine $A = a_0a_1a_2...a_m$, $B = b_0b_1b_2....b_n$, and $C = c_0c_1c_2...c_{m+n}$, where $a_0 = b_0 = c_0 = \epsilon$ (ϵ is the empty string). For $i, j \in \{0,1,...,m\} \times \{0,1,...,n\}$, define $M(i,j)$ to be the boolean value that represents true *if and only if* $A_i := a_0a_1a_2...a_i$ and $B_j := b_0b_1b_2....b_j$ are mutually involved in $C_{i+j} := c_0c_1c_2...c_{i+j}$. The idea is that if A_i and B_j are mutually involved in C_{i+j}, then c_{i+j} has to be either a_i or b_j. So to evaluate $M(i,j)$,

[2] like before another way that helps is to try to see the solution as a bottom top approach, thus starting from the leaves and building our way up, thus we notice that over every edge $\{u, v\}$ we will do a $O(1)$ number of computation (one for each of the $OPT(u), OPT(u_{\text{rest}}), OPT'(u)$ calls and a constant number of arithmatic operations at the parent node v) until we find the final solution when at the root. Thus, the total cost is the sum of all what we described that would happen over every edge and since we have $O(n)$ edges in a tree, we get an $O(n)$ running time.

we first check if this condition is satisfied: if it is not, then we can directly decide that $M(i,j) = \texttt{false}$; otherwise, we then accordingly recurse on the remaining part of C_{i+j-1} and check whether A_{i-1} and B_j or A_i and B_{j-1} are mutually involved in C_{i+j-1}, and if so we can finally decide that $M(i,j) = \texttt{true}$, else it is false.
We thus obtain the following recursion:

$$M(i,j) = \begin{cases} \texttt{true} & \text{if } i = j = 0 \\ M(i-1,j) & \text{if } a_i = c_{i+j} \wedge b_j \neq c_{i+j} \\ M(i,j-1) & \text{if } a_i \neq c_{i+j} \wedge b_j = c_{i+j} \\ M(i-1,j) \vee M(i,j-1) & \text{if } a_i = b_j = c_{i+j} \\ \texttt{false} & \text{if } a_i \neq c_{i+j} \wedge b_j \neq c_{i+j} \end{cases}$$

Based on that, we propose the following algorithm using memoization:

Algorithm 3 `Mutually-Involved`(m, n) ▷ `memo` is a 2-dimensional array initially empty

```
if $m = n = 0$ then return true                                          ▷ base case
if memo$[m, n] \neq$ Null then return memo$[m, n]$                       ▷ decision was made before
memo$[m, n] \leftarrow \bigvee \begin{cases} \texttt{Mutually-Involved}(m-1, n) & \text{if } a_m = c_{m+n} \\ \texttt{Mutually-Involved}(m, n-1) & \text{if } b_n = c_{m+n} \\ \texttt{false} & \text{otherwise} \end{cases}$     ▷ Memoization
return memo$[m, n]$
```

Running time: we have $(m+1) \times (n+1)$ entries in our 2D array, i.e. $(m+1) \times (n+1)$ subproblems. Notice that when filling the array and due to memoization, we compute each entry value $\texttt{memo}[i,j]$ for $i, j \in \{0, 1, ..., m\} \times \{0, 1, ..., n\}$ at most once. Also, each computation of $\texttt{memo}[i,j]$ costs $\mathcal{O}(1)$ in the current step (i.e. not counting the cost of recursive calls, thus the only thing we are doing is determining whether the *if* clauses are satisfied and then evaluating the $\vee$ operator). Therefore, the total running time is $\mathcal{O}(m \cdot n)$.