CS301 HW4

Adnan Kaan Ekiz

May 2019

1 Problem 1

1.1 Part A

1.1.1 Recursive Algorithm

For the recursive formulation of LCS problem with k different strings, we can start with the last letter of all of the k strings. If we find out that all k strings have the same last letter, then length of the largest common subsequence will increase by 1 and algorithm again needs to call the same function for k strings after excluding the last element of the strings.

However, if all strings does not consist the same element, then algorithm recursively calls the function for all strings after excluding the last letter of the specific string that is called for. Example pseudocode of the algorithm for k=3 strings is given below:

Algorithm 1 LCS Algorithm For 3 Strings

```
1: procedure LCSFINDER [STRING A, STRING B, STRING C, INT LENA, INT LENB, INT LENC]
2:     if Lengths has an element 0 then
3:         Return 0
4:     if all k strings has the same last element then
5:         Return LCSfinder [A, B, C, lenA-1, lenB-1, lenC-1] + 1
6:     else
7:         Return Max(LCSfinder [A, B, C, lenA-1, lenB, lenC],
8:         LCSfinder [A, B, C, lenA, lenB-1, lenC],
9:         LCSfinder [A, B, C, lenA, lenB, lenC-1])
```

This algorithm can be applied for k>3 strings by only changing small part of the code. Depending on the number k, algorithm will have the complexity $O(k^n)$, which is exponential.

1.1.2 Top-Down Algorithm

For the top-down algorithm, only difference with the recursive problem will be adding an extra memory table in the form of dictionary. At each call of the function itself, algorithm will check if the given problem is in the memory table. If it is in the memory table then algorithm will retrieve the data from the memory table. If it is not in the memory table algorithm will compute the function and add it to the memory table.

Top-down algorithm can also be applied for k>3 strings by only changing small part of the code. Depending on the number k, algorithm will have the complexity $O(k * n)$ where k is the number of strings and n is the length of the string. Pseudocode of the algorithm would be:

Algorithm 2 LCS Top-Down Algorithm For 3 Strings

```
procedure LCSFINDER [STR A,STR B,STR C,INT LENA,INT LENB,INT LENC,MEMORY TABLE]
    if Lengths has an element 0 then
        Return 0
    if all k strings has the same last element then
        Return LCSfinder [A, B, C, lenA-1, lenB-1, lenC-1] + 1
    else
        if step is calculated before then
            Return Table[current]
        else
            $value \leftarrow Max(LCSfinder[A, B, C, lenA - 1, lenB, lenC],$
                LCSfinder [A, B, C, lenA, lenB-1, lenC],
                LCSfinder [A, B, C, lenA, lenB, lenC-1])
                Table[current] ← value
                    Return value
```

1.2 B

Asymptotic time complexity of the recursive algorithm will be $O(k^n)$ where k is the number of strings and n is the maximum string length. At each call of the function, algorithm will call the same function for every length of the given strings for at most n times. Given that total string number is k, then for the last call it means there has been at most k^n call. Hence, complexity is $O(k^n)$ for recursive algorithm.

For the Top-Down algorithm, there will be at most 1 call for given parameters in the algorithm. Because of the memoization, there can be at most $O(k * n)$ calls where each call is calculated only once, Hence total complexity is $O(n)$ which is linear time.

It is expected since memoization technique keeps the computed information so that recursive function does not have to compute the same problem over and over again. Meantime, recursive function computes the same problem for more than once since value is not kept in the memory table, resulting in exponential time.

2 Problem 2

2.1 A

Algorithm will take an empty array and a root pointer as a parameter for the recursive function. For the description of the algorithm, subproblems are the following:

- Algorithm checks whether given node is null or not. If it is null, then returns 0.
- Algorithm checks if current node's children are null. If both of the children of current node are null, then returns 0.
- Calculates size of vertex cover when root is part of it and adds the id of the root to set when root is part of it.
- Calculate size of vertex cover when root is not part of it and adds the id of the root to the set when root is part of it.
- Returns the minimum of two sizes since we need the minimum amount elements in a given set. Then uses the set that belongs to the minimum size.

For the given description pseudocode of the algorithm becomes the following:

Algorithm 3 Minimum Cardinality Set

```
procedure VC [NODE *ROOT, ARRAY SET, MEMORY MEM]
    if root is NULL then return set
    if left and right child are NULL then return set
    if setMem[current node] exists then return setMem[current node]
    set1 ← set
    set2 ← set
    set1 ← add(root.id)
    set2 ← add(root.id)
    sizeInc ← 1 + VC[root.left, set1,mem] + VC[root.right, set1,mem]
    sizeExc ← 0
    if root.left then
        sizeExc ← sizeExc + 1 + VC[root.left.left, set2, mem].length
        + VC[root.left.right, set2, mem].length
    if root.right then
        sizeExc ← sizeExc + 1 + VC[root.right.left, set2, mem].length
        + VC[root.right.right, set2, mem].length
    if sizeInc < sizeExc then
        return set1
    else
        return set2
```

2.2 B

For the algorithm given above, since algorithm uses the same function and all the data computed will be stored in the memory table called mem. For that reason, algorithm calculates each node in the tree for only once. Therefore algorithm has the complexity of $O(n)$ where n denotes the total number of nodes.

2.3 C

Some problems are extremely unlikely to have a polynomial time algorithm, even for planar graph. But the same problem might be polynomially solvable on trees using dynamic programming. If we want some algorithm to be efficient then we need to bound the number of states in the dynamic programming table. This leads to following reasons:

- For a polynomial-time algorithm, then we need a polynomial number of states.
- Number of states typically corresponds to the number of different types of partial solutions when breaking the graph at some small separator.
- Thus a problem is easy on bounded-treewidth graphs(trees) usually because partial solutions interacting with the outside world via a bounded number of vertices have only a bounded number of types.

Therefore, because of the reasons mentioned earlier, trees will create a bound for the problem to solve which limits the possible solutions hence decreasing the asymptotic time to polynomial time instead of exponential time in general. Of course not every problem can be changed to polynomial asymptotic time, but for some problems it is enough to make a huge difference in the time complexity of the algorithm.

3 Problem 3

3.1 A

A leaf vertex in a tree is considered as a vertex with degree 1 in a tree where degree of vertex is equal to the number of edges that contained by the vertex. A minimum leaf spanning tree is a problem that given a graph G = (V,E) and an integer i, checks if there is any spanning tree T in G that contains at most i leaves. So a minimum leaf spanning tree is a spanning tree with the minimum number of vertices with degree 1. Given this information input and output of the problem would be:

- Input: Graph G = (V,E) and Integer i
- Output: Subgraph T = $(V^{'}, E^{'})$ where $V^{'}$ is the set of vertices such that $V^{'} \subseteq V$ and $E^{'}$ = E.

3.2 B

Real world application of this problem would be similar to Hamiltonian Path Problem since in Hamiltonian Path, main objective is to visit all vertices without even denying any single vertex.

When these important points considered, real application of the problem can be distribution networks for any given product or it even can include laying out electrical grids since each of these problems will have a specific starting point in the network. Then from these starting points, real motivation would be to distribute a product or resource we have to every given point in that network. This problem also include the importance of using roads or wires as less as possible since each extra transportation between and given edge will have extra cost. In addition to that, minimum number of leaves will be considered since there isn't any connection from any given leaf to another vertex except the vertex that is connected to the leaf which is already used in the process of reacting to the leaf.

3.3 C

To prove that a problem is NP-complete, we must first guess a solution non-deterministically. Afterwards, we must come up with an algorithm that checks whether the given solution will satisfy the problem or not. After that, algorithm must be in polynomial time to prove that the problem is NP-complete. According to this definition proof of this problem being NP-complete would be:

Algorithm 4 Minimum Leaf Spanning Tree

```
procedure MLST [GRAPH G, INT i, TREE T]
    count ← 0
    for each $v \in T$ do
        if v.degree = 1 then
            count ← count + 1
        if count > i then
            Return false
    Return true
```

- Non-deterministically guess a solution for all e ∈ E, choose nondeterministically if e is to be included in T.
- Came up with an algorithm that checks whether the given problem can be proved in polynomial time. For the pseudocode displayed in above, total complexity of the algorithm is $O(V)$ hence in fact it has a total asymptotic complexity of $O(n)$ which is polynomial.

- Therefore, it is possible to state that this proof of the solution is in polynomial time which makes the problem NP-complete.