

CS 301 – Algorithms

Homework 4 – 13/05/2019

Atilla Alpay Nalcaci - 19510

Problem 1. Longest Common Subsequence






k sequences are given as input and the goal is to find their LCS ($k > 2$).

(a) For this problem, the longest common subsequence problem is modified as follows;

- After each word is added to array, the LCS_MULTIPLE_SEQ functions as comparing 2 strings for each loop, until it reaches the end of array.
- Once a comparison of two strings complete, the program replaces the value of the first string (a.k.a the left string) with the string value that it produced (the string value which holds the LCS of these two strings) and removes the second string (a.k.a the right string).
- It terminates once there is no more than one string left.
- With this principle, the program is bound to find the LCS of $k > 2$ sequences since, by storing the LCS of two strings each time, it ultimately searches for the common letters which are common in all strings.

Followingly, pseudocode representation of the above algorithm provided below (in two sections);

```

LCS_WORD(s1, s2):  Returns the LCS; S1 signifies rows,
    rows = len(s1) + 1  S2 signifies columns
    cols = len(s2) + 1
    dp_array = [[0 for i in range(cols)] for j in range(rows)]
     Row0 and col0 is zero since they signify null strings
    for i in range(1, rows):
        for j in range(1, cols):
            if s1[i - 1] == s2[j - 1]:  If characters match, top-left diagonal value + 1
                dp_array[i][j] = dp_array[i - 1][j - 1] + 1
            else:  Else, compare top and left values, get the maximum
                dp_array[i][j] = max(dp_array[i - 1][j], dp_array[i][j - 1])

```

```

sub_sequence = [] → Store the LCS in reverse order

i = rows - 1
j = cols - 1

while i > 0 and j > 0:
    if dp_array[i][j] != dp_array[i - 1][j]: → If the top value is not equal, it means that the
        sub_sequence.append(s1[i - 1])           present character was used
        i -= 1
        j -= 1
    else: → Else, it is not used
        i -= 1

return sub_sequence[::-1] → Return the list of LCS

```

Figure 1. Longest Common Subsequence Pseudocode

```

LCS_MULTIPLE_SEQ(arr): → Returns the LCS of k>2 sequences (arr = [])

i = 0
if len(arr) % 2 == 0:
    while i < len(arr):
        if i + 1 < len(arr) - 1 or len(arr) % 2 == 0:
            if i + 1 < len(arr):
                m = len(arr[i])
                n = len(arr[i + 1])
                k = LCS_WORD(arr[i], arr[i + 1])
                a = ""

                for z in range(len(k)):
                    a = a + k[z]

                arr[i] = a
                k = arr.pop(i + 1)
                i += 1

            else:
                break
        else:
            arr[i] = arr[i + 1]
            k = arr.pop(i + 1)
            i = 0
    else:
        while i < len(arr):
            if i + 1 < len(arr) - 1 or len(arr) % 2 == 0:
                if i + 1 < len(arr):
                    m = len(arr[i])

```

```

        n = len(arr[i + 1])
        k = LCS_WORD(arr[i], arr[i + 1])
        a = ""
        for z in range(len(k)):
            a = a + k[z]
        arr[i] = a
        i += 1
    else:
        break
else:
    arr[i] = arr[i + 1]
    k = arr.pop(i + 1)
    i = 0
k = arr.pop(1)
return arr[0]

```

Figure 2. *Multiple Longest Common Subsequence Pseudocode*

(b) For this problem, asymptotic time complexity is the size of the array (input array) multiplied by the square of the length of longest string in the array:

$$\text{Time} = \text{len}(\text{arr}) * (\text{len}(\text{arr}[\text{c}]))^2$$

[arr: Input Array; c: Slot in arr that has the longest string]

Problem 2. Vertex Cover Problem

(a) Given problem is described as a mathematical discipline of graph theory, referred as *Vertex Cover*, where a set of vertices (which is X in this case) for a given graph $G = (V, E)$, each edge of the graph in this set is incident to at least one vertex of X .

Thus, problem indicates to finding a solution to the following question: *For a graph $G = (V, E)$ and subset of vertices X , find the minimum size vertex cover where for all $(u, v) \in X$, either $u \in X$ or $v \in X$.*

Additionally, the solution must consider whether if the chosen root is part of vertex cover or not.

- If YES then recursively calculate the size of vertex covers for left and right subtrees and add to the result in order to include root.
- If NO then recursively calculate the size of vertex covers of all grandchildren and number of children to the result (since both children of root must be included in vertex cover in order to cover all root to children edges).

So a dynamic programming approach to the problem would be as;

- i. Determine a root arbitrarily ($v \in V$ and $G = (V, E)$)
- ii. Identify subproblems (for $v \in V$: size of smallest vertex cover in subtree rooted at v)
- iii. Is v in the cover?
 - If YES, then cover the adjacent edges.
 - If NO, then all adjacents must be in the cover.

Followingly, a recursive solution would be as follows;

$$V(v) = \min(\text{sum}(V(c)+1, \text{sum}(V(g) + \text{len}(v.\text{children}))$$

[V: Vertex Cover Function; v: Root Node; c, g: Arbitrary Nodes]

Note that the Vertex Cover problem sustains the both properties of a dynamic programming problem since same problems are being calculated more than once (e.g. same nodes being processed multiple times) and an optimal solution is obtainable by the optimal solutions of its subproblems (e.g. the vertex cover of X is obtainable by the vertex cover of its subtrees), Overlapping Subproblems and Optimal Substructure properties respectively.

Pseudocode of the algorithm is provided below:

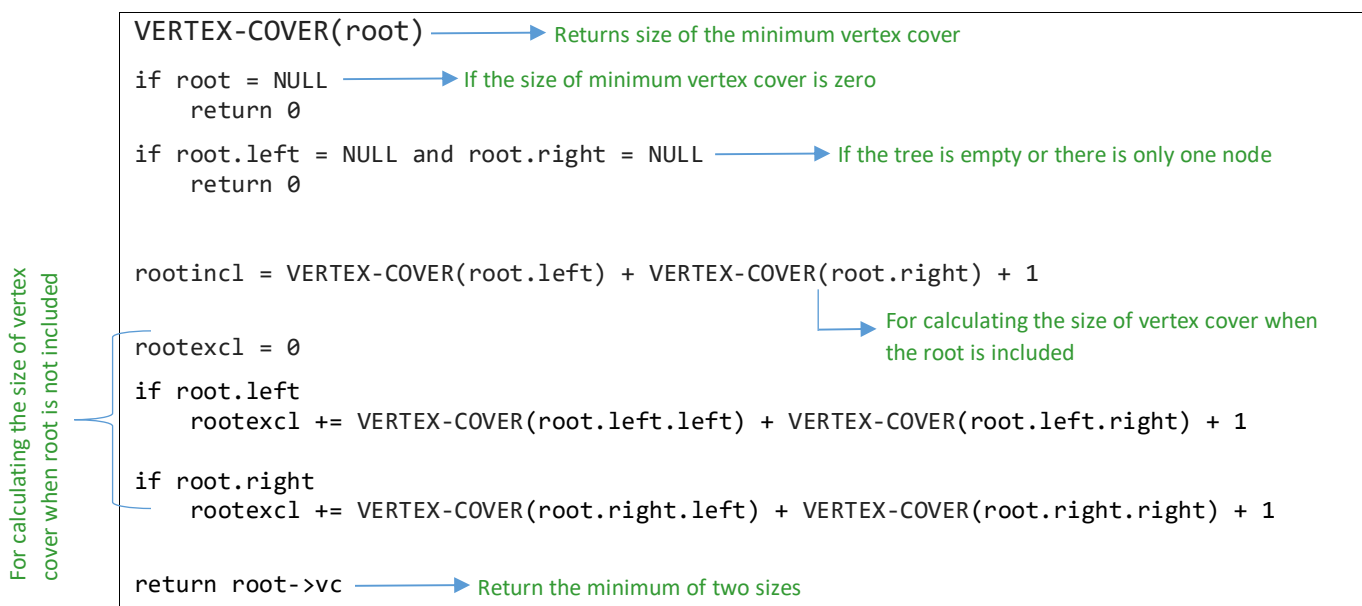


Figure 3. Vertex Cover Problem Pseudocode

(b) For this problem, the asymptotical time complexity can be analyzed as the multiplication of number of subproblems with the time it takes to travel all vertex, each divided by the total time of their subproblem:

$$\text{Time} = \# \text{ of subproblems} \cdot (\text{time} / \text{subproblem})$$

$$\rightarrow |V| \cdot O(V) = O(V^2)$$

→ Actually $O(V)$ since each vertex visited twice

(c) Computational complexity would have decreased if a graph is given as an input instead of a tree since each vertex would have been visited twice if the input was a graph. Via dynamic programming algorithm that is provided in **part (a)**, this problem can be solved and the actual complexity would be $O(V)$ in that case.

Problem 3. Minimum Leaf Spanning Tree

(a) The *Minimum Leaf Spanning Tree* (MLST) is a problem that, for a graph $G = (V, E)$ and an integer k , is there a spanning tree X in G that contains at most k leaves. As it can be observed from its structure, it is a decision problem for finding whether such spanning tree exists.

Input: an undirected graph G and a positive integer k

Output: “decide” whether a spanning tree with at most k leaves exists

(b) A real-world application of MLST can be given as the design of network systems. For instance, a telecommunications company laying cable to certain area. If the design is constrained to bury the cable only along certain paths (e.g. along roads), then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive (requiring cable with greater length) thus these paths being represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, thus would represent the least expensive path for laying the cable.

(c) To prove that a given problem is NP-complete, we must proceed along the following steps;

- i. Guess a solution for the given problem
- ii. Implement a pseudocode based on this solution
- iii. Compute the complexity of the solution.

In general, the problem is in NP since the tree itself is a polynomial witness for membership in this language. The restricted version of this problem where $k = 2$ is equivalent to the NP-complete *Hamiltonian Path Problem*.

Hamiltonian Path Problem

Hamilton Path problem is described as a decision problem for determining whether, for a graph $G = (V, E)$, there exists a path that visits each vertex exactly once. Basically, a Hamiltonian Path is a simple open path that contains each vertex in a graph exactly once and the problem determines whether there exists a Hamiltonian Path or not (for a graph G).

To prove that this problem is in NP, we must show that it belongs to class NP (as in polynomial time) then find a known NP-complete problem that can be reduced to our problem.

1. Hamiltonian Path chooses edges for a given graph G nondeterministically which are to be included in the path and traverses the path while making sure that we visit each vertex exactly once. The choosing and traversing processes can obviously be done in polynomial time, making the problem belong to NP.
2. A fairly known NP-complete problem that can be reduced to prove that Hamiltonian Path problem is in NP is *Hamiltonian Cycle* where a Hamiltonian Path beginning and ending in the same vertex.

Proof is as follows; given a graph $G = (V, E)$, construct a graph G_0 where G contains a Hamiltonian Cycle if and only if G_0 contains a Hamiltonian Path. This is done by choosing

an arbitrary vertex u in G and adding a copy of it (u') together with all its edges. Finally, add vertices v and v' to the graph and connect v with u and v' with u' .

Suppose G contains a Hamiltonian Cycle. Then G_0 consists of a Hamiltonian Path that starts from v and following the cycle from G back to u' , ending at v' . Conversely, suppose G_0 contains a Hamiltonian Path. The path must necessarily have endpoints in v and v' . This path can be transformed to a cycle in G if we disregard v and v' . The path must have endpoints in u and u' and by removing u' we get a cycle in G the path back to u is closed.

If G has a single edge, the conditions do not work, so it must be taken care of as a special case. Ultimately, G contains a Hamiltonian Cycle if and only if G_0 contains a Hamiltonian Path, which concludes the proof that Hamiltonian Path is NP-complete which also concludes that Minimum Leaf Spanning Tree is also in NP.