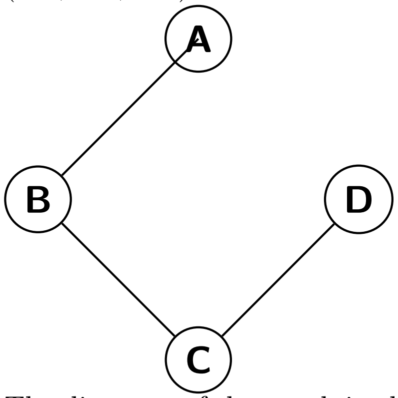
CSOR W4246—Fall 2019

Name: Prasham Dhaneshbhai Sheth
UNI: pds2136

HW2—Theoretical part

1. **Solution to problem 1**

   (a) i. Following figure shows a graph G consisting of 4 nodes (A, B, C, D) and 3 Edges (AB, BC, CD).



The diameter of the graph in this case is the maximum of distance between any two nodes (i.e)
$diam(G) = max(distance(A,B), distance(A,C), distance(A,D),$
$distance(B,C), distance(B,D), distance(C,D))$

$$diam(G) = max(1,2,3,1,2,1)$$

$$diam(G) = 3$$

$apd(G) = \frac{distance(A,B)+distance(A,C)+distance(A,D)+distance(B,C)+distance(B,D)+distance(C,D)}{\binom{n}{2}}$
$apd(G) = \frac{(1+2+3+1+2+1)}{\binom{4}{2}}$
$apd(G) = \frac{10}{6}$
$apd(G) = \frac{5}{3}$
Thus, $diam(G) \neq apd(G)$

ii. By definition, Diameter of the graph is longest of the shortest paths between all pairs of the nodes (i.e)

$$diam(G) = max(distance(u,v));\ (u,v) \in V$$

and average pairwise distance in G (i.e) apd(G) is

$$apd(G) = \frac{\Sigma distance(u,v)}{\binom{n}{2}};\ (u,v) \in V$$

Thus, we can say that the average is summation of all pairwise distances divided by number of edges in an undirected graph. By the formula it can be said that

$$diam(G) \geqslant apd(G)$$

The above inequality holds due to the fact that average of n quantities is summation of all of the n quantities(which at maximum can be n * maximum of all quantities: the case when all the quantities are equal) divided by number of the quantities that is n.
In our case quantities are the pairwise distances and there are $\binom{n}{2}$ such distances.

$$\frac{diam(G)}{apd(G} \geqslant 1$$

This gives the lower bound for the inequality.

Now lets assume that the given inequality $\frac{diam(G)}{apd(G)} \leqslant c$ holds true for certain c = c1
Now consider a graph G1 having n nodes for which the above inequality holds true. The fact that the above inequality shall hold true for a constant value of c makes it invalid as if we add k nodes in the direction of the diameter making the diameter increase by k, we effectively are increasing the diameter by k. The apd won't increase by k as the increased pairwise distance is divided by increased number of nodes. We always can find such a value of k using which we can get a graph for which the above ratio of diameter and average pairwise distance is greater than c1. This contradicts the assumption that inequality is true for all graphs for a fixed positive value of c. Thus, we can say that even though there can be a case that the above ratio is less than c for a fixed Graph we cannot generalize the inequality for that fixed c for all the graphs.

(b) The following is the algorithm to compute the diameter of the tree. The algorithm utilizes the Depth First Search algorithm to find the deepest node and then Depth First Search from that node is performed to get the diameter of the tree. The idea follows the intuition that in any tree the longest path would start from the deepest leaf and go to another leaf. By the first call of DFS we find the deepest node in the tree and then using that node we apply DFS again to get the deepest node from there and the length of path to that node is the diameter of the tree. This is evident from the observation that in any undirected tree the longest path is the between two leaves only.

Algorithm 1 Algorithm to compute diameter of an undirected tree

```
Compute_Diameter(G = (V,E)):
1: u = root of the Tree
2: node_max_depth,depth = DFS (G, u)
3: node, diameter = DFS(G, node_max_depth)
4: return diameter

DFS(G = (V,E) , s):
1: for $u \in V$ do
2:     explored[u]=0
3:     distance[u]=0
4: end for
5: count=0
6: distance = Search(s,count,distance)
7: Find node x with maximum distance using distance array
8: return x, maximum distance

Search(u,count, distance):
1: if $distance[u] < count$ then
2:     distance[u] = count
3: end if
4:  explored[u] =1
5: for $(u, v) \in E$ do
6:     if explored[v] == 0 then
7:         distance = Search(v,count+1,distance)
8:     end if
9: end for
```

Correctness:

The correctness for the algorithm can be proved using induction.

Base case: consider a graph with a single node. trivially the diameter of the graph is zero and our algorithm would run DFS from the same node and would return 0 as required.

Hypothesis: Suppose the algorithm runs correctly for a tree having n nodes.

Step: For a tree with (n+1) nodes, the (n+1) th node can thought to appended to any leaf node of a tree with n nodes.

As our algorithm correctly estimates the diameter of a n node tree, we can say that it can correctly identify the deepest node for a tree of size n. Adding a node can make the deepest node of the previous tree to become the second deepest node or still be the deepest node. in the case it still is the deepest node from the hypothesis we can say that it will correctly output the correct diameter of the graph as we know that using DFS we would definitely reach the deepest node(as each an every child of the node would be explored). Similarly if the new added node is the deepest node; our algorithm would correctly estimate it as the deepest node as the first call to the DFS would estimate the correct depth of all nodes as it explores all the nodes of the tree.

Hence, we can say that our algorithm works for a tree with n+1 nodes as well. And hence, we can prove the correctness of the given algorithm.

Time Complexity:

The algorithm calls DFS 2 times and DFS is a O(V+E) and hence the complexity of the given algorithm is also O(V+E).

Space Complexity: The given algorithm requires to maintain arrays of size V for both distance and marking the node as visited. Thus, it is having O(V) space complexity.

2. **Solution to problem 2** The example requires to compute the shortest path to all vertices from a given source vertex with keeping into consideration the cost of visiting a vertex as well as the cost corresponding to the edge in the path. The task is similar to Dijkstra's algorithm with an added constraint of considering the vertex cost as well. The following algorithm is an updated version of the v2 algorithm taught in the class for finding the shortest path using Dijkstra's algorithm.

Algorithm 2 Algorithm to compute distance of vertices from the source in a directed graph. Function takes Graph G = (V,E, w, c) as input and outputs an array of distances.

```
Compute_Distances(G = (V,E,w,c), $s \in V$):
1: Initialize(G, s, c)
2: $S = \phi$
3: while $S \neq V$ do
4:     Pick u so that dist[u] is minimum among all nodes in V-S
5:     $S = S \cup \{u\}$
6:     for $(u, v) \in E$ do
7:         Update(u,v)
8:     end for
9: end while
Update(u,v):
1: if $dist[v] > dist[u] + w_{uv} + c_v$ then
2:     $dist[v] = dist[u] + w_{uv} + c_v$
3:     $prev[v] = u$
4: end if
Initialize(G, s, c):
1: for $v \in V$ do
2:     $dist[v] = \infty$
3:     $prev[v] = NIL$
4: end for
5: $dist[s] = c_s$
```

Correctness:

The algorithm is a modified version of the Dijkstra's algorithm taught in the class and the correctness of the algorithm follows the correctness of Dijkstra's Algorithm taught in the class. The algorithm can be also thought of as the same Dijkstra' algorithm with an updated weight of edge cost. The cost of edge is updated by adding the weights of vertex cost to each edge of that incoming edge. Also, the starting cost is initialized as the vertex cost of the source node.

Time Complexity:

The time complexity of the given algorithm is same as that of Dijkstra's algorithm which is implemented here using an array and hence has a time complexity of O(n^2). The tie complexity can be improved by using a priority queue implemented as a binary min-heap.

Space Analysis: The given algorithm is having O(V) space complexity as it consumes linear amount of space for holding the set of vertices and two arrays of size V to store distances and to store the previous node for the shortest path.

3. **Solution to problem 3**

The intuition behind designing the following algorithm (which checks whether the given string can be split into a set of valid dictionary words separated by spaces and if it is, it returns the sequence of valid words that constitute the string) is to check whether the substring (1..i) is a valid split-able string or not. This is checked for all values of i ranging from 1 to n. if a substring (1..j) is splitable, our problem reduces to identifying whether the substring (j+1..n) is splitable or not. Thus a recursive check is possible but it would lead to solving overlapping sub-problems and hence the recursive solution is converted to a Dynamic Programming one to reduce the time consumption at a cost of some extra space. The array holds the value showing whether the spit is possible or not. The value of m[i] = 1 id substring [1..i] is validly splitable and 0 otherwise.

The recursive equation for the following question is

$OPT(i,j) = ((OPT(i,i)\ and\ OPT(i+1,j))\ or\ (OPT(i,k)\ and\ OPT(k+1,j)))\ ;\ for\ all\ i < k < j$

OPT(m,n) = 1 if substring [m..n] is in in the dictionary ; 0 otherwise

Thus, we keep a single dimensional array for storing the values and it is filled from start to end.

Correctness:

We can prove the correctness of the algorithm by using Induction.

Base case: When the given string is of one character the algorithm correctly outputs whether the given character is in the dictionary forming a word or not.

Hypothesis: Assume that the algorithm runs correctly for n sized string and correctly outputs the Boolean representing the existence of valid split of the string and if it is valid it returns the set of words in which the string can be space separated.

Induction: If we take a string of size (n+1). Using the hypothesis we know that the algorithm correctly identifies whether or not the string can be splitted into a set of valid dictionary words or not. For the (n+1) length string, we return that the string is splitable into space separated dictionary words or not by examining the value of m[n]. The for loop inside the if condition check the validity of each substring from (i+1)th character to the last character for validity of it as a dictionary word. If it finds that the word is valid, it will set the value for that the

Algorithm 3 Algorithm to determine if the string S can be reconstituted as a sequence of valid words and outputs the corresponding sequence of words, if the string is valid. In the case when string is not breakable into valid dictionary words it return False

```
String_Validity(S):
m [] = Array of size n initialized by 0
Flag=0
for i=1 upto n do
    if $m[i] == 0$ and $dict(s[1..i]) == 1$ then
        m[i]=1
        Flag = 1
    end if
    if $m[i] == 1$ then
        if i != n then
            for j = i+1 to n do
                if m[j] == 0 and dict(s[i+1..j]) == 1 then
                    m[j]=1
                    Flag=1
                end if
                if j== n and m[j]==1 then
                    break_value = 1
                    Break the loop
                end if
            end for
            if break_value==1 then
                Break the loop
            end if
        else
            Flag=1
        end if
    end if
end for
if Flag == 0 then
    return False
else
     j=1
    for i=1 upto n  do
        if m[i]==1 then
            Output s[j..i]
            j=i
        end if
    end for
end if
```

corresponding index as 1. Hence, the inner loop takes into consideration the new updated length of the string and hence correctly outputs for the (n+1) length of string(this can be said as the hypothesis hold true and we know the loop performs perfectly and as the loop counter has length(string) into consideration, it will definitely work for string of length n+1)

Hence, we can say that the algorithm runs correctly.

Time Complexity:

Lines 1 and 2 runs for single time and hence are performed in O(1) time. The lines 4-9 inside the for loop runs the same number of times the loop runs. The particular loop on line 3 runs for n times. The lines inside inner for loop runs for (n-i) times (i.e) O(n). Thus, overall when the outer loop is run it takes O(n^2) time. Hence, the given algorithm has O(n^2) running time complexity.

Space Analysis: The given algorithm is having O(n) space complexity as it consumes linear amount of space for holding the array (of 0s and 1s) having its size proportional to the size of given string.

4. **Solution to problem 4**

   The problem requires us to identify whether the first string is the subsequence of another or not. We here can test whether the first string is a subsequence of second by assuring that all the characters of the first string appears in the second one in a sequential order. For this what I did was that I traversed the second string and for all the characters of the first string the counter was incremented which at the end of the traversal is compared with the length of first string. If the count and length of the string are equal, we can say that first string is the subsequence of second string. if that is not the case we can return False as first string won't be a subsequence of the second one.

Algorithm 4 Algorithm to find whether S' is a sub-sequence of S:
The function returns True if first string s' is a sub-sequence of second string s, otherwise returns False.

```
isSubsequence(s', s):
count = 0
if length(s') == 0 then
    return True
end if
if length(s') > length(s) then
    return False
end if
for  i=1 up to length(s) do
    if (s'[count] == s[i]) then
        count = count + 1
    end if
    if (count == length(s') + 1) then return True
    end if
end for
return False
```

Correctness:

We can prove the correctness of the algorithm by using Induction.

Base case: if the first string is of 0 length, it by default is the subsequence of the second string and hence our algorithm should return true which it does by checking whether the length of first string is 0 or not.

Loop Indeterminate: After every loop i the count holds the value of index that is to be compared next with the i-th character of the second string. The previous characters in the first string that the characters before the (count)th character have already appeared and hence if at any time the count exceeds the length of string, all the characters of the first string must have appeared in the second string and hence the loop terminates by returning True.

Loop termination: The loop is for loop and it would terminate when either all the characters of the first string have appeared in the second string or else the iterator variable i exceeds the length of second substring. In the first case the if condition inside the loop is satisfied and hence the loop terminates and True value is returned. In the later case all the characters of the first string won't be present in the second string and hence we can say that the algorithm must return false which it does after the loop terminates that way.

Hence, the loop always terminates and the algorithm return correct value in each case and so we can say that the algorithm would always give correct output.

Time Complexity:

The first 4 lines of the algorithm and last line of the algorithm run only once, the lines inside the for loop are ran (length of second string) times in the worst case. Thus, we can say that the given algorithm is having O(n) time complexity.

Space Analysis: The given algorithm is having O(1) space complexity as it consumes constant space for any inputs.

5. **Solution to problem 5** The task that need to be performed by this algorithm is to minimize the effort of combining the stacks. For this we divide the problem into subproblems of combining a subset of stacks and then combining the subset with the remaining stacks. The recursive equation of the algorithm is give below. The main intuition behind the solution is that if we optimally combine all subproblems having fewer stacks, the main problem of optimally combining all the stacks will be solved.

   The recursive equation for solving the question can be written as:

$$OPT(1,n) = min\{OPT(1,K^*)+OPT(K^*+1,n)+height(1,K^*)+height(K^*+1,n)\}; 1 <= K^* <= n$$

   Here, height(p,q) represents the height of the resultant stack after combining stacks from pth index to qth index. Height(p,q) is equal to the summation of heights of all stacks between p and q index value.

   The boundary condition for the given recursive logic is that OPT(i,i)=0 (i.e) effort of combining a single stack is 0.

Algorithm 5 Algorithm to compute the minimum effort required by Alice to merge all her books in a single stack

Function return the minimum amount of effort required by Alice to perform the task

```
minimumEffort(height):
n = length(height)
Height_Step [ ][ ] = Array (n x n)          ▷ Initialize a 2D array of size n*n
Effort [ ][ ] = Array (n x n)               ▷ Initialize a 2D array of size n*n
for i=1 up to n do
    Height_Step[i][i] = height[i]
end for
for m = 1 to n do
    for i = 1 to (n - m) do
        Height_Step[i][i+m] = Height_Step[i][i] + Height_Step[i+1][i+m]
    end for
end for
for i =1 to n do
    Effort[i][i] = 0
end for
for m = 1 to n do
    for i = 1 to (n - m) do
        j = i+m
        minimum = + ∞
        for  k = i to j-1 do
            value = Height_Step[i][k] + Height_Step[k+1][j] + Effort[i][k] + Effort [k+1][j]
            if minimum > value then
                minimum = value
            end if
        end for
        Effort[i][j] = minimum
    end for
end for
return Effort [1][n]
```

In the given algorithm the array Effort stores the effort that would be required to combine the stacks and value of Effort[i][j] represents the cost of combining towers from index i to j. Height_Step[i][j] stores the resultant height after combining stack i to j.

Time Complexity:

The time complexity of the given algorithm is O(n^3). This is evident from the fact that there are n^2 subproblems and each subproblem requires O(n) time for computation(provided we know the solutions for all subproblems).

Space Analysis: The given algorithm is having O(n^2) space complexity as it stores 2 n * n array for storing cumulative height and the values for effort of combining the stacks.

Order of solving: The algorithm follows a diagonal by diagonal way of solving. The primary diagonal represents problem of 1 stack and it goes on increasing as we go towards the upper diagonal. The corner most element of the array represents the problem of n stacks which is the problem we are required to solve.