FOUNDATIONS OF ALGORITHMS – CSE 551 – ASSIGNMENT 3

1.

Consider the four matrices that have to multiplied be P, Q, R and S.

Given, the dimensions of the four matrices P, Q, R and S as 13x5, 5x89, 89x3 and 3x34 respectively.

Using dynamic programming as discussed in class, the number of multiplications needed for the multiplication of P, Q, R and S can be calculated using the following equation:

$$m[i,j] = \begin{cases} 0, & \text{if } i=j \\ \min_{i \le k \le j} [m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j] \end{cases}$$

Here, m[i,j] gives the minimum number of multiplications required to multiply the matrices from i to j.

Hence, in this particular problem, m[1,4] gives us the final result of minimum number of multiplications for multiplying the four matrices.

Now, using the above equation, we get

m[1,1] = m[2,2] = m[3,3] = m[4,4] = 0

Putting these values into a i and j table , we get the below table:

j

1	0			
2		0		
3			0	
4				0
	1	2	3	4

i

Now, we know that the number of multiplications required to multiply two matrices of dimensions axb and bxc is a.b.c

Hence,

m[1,2] = 13.5.89 = 5785

m[2,3] = 5.89.3 = 1335

m[3,4] = 89.3.34 = 9078

After inserting these values to the above table, we get:

1	0			
2	5785	0		
3		1335	0	
4			9078	0
	1	2	3	4

j

i

Using the initial equation of m[i,j] for calculating the remaining values,

m[1,3] = min(m[1,1] + m[2,3] + 13.5.3, m[1,2] + m[3,3] + 13.89.3)

m[1,3] = min(0 + 1335 + 195, 5785 + 0 + 3471) = min(1530, 9256) = 1530

This minimum of m[1,3] is obtained using 1 as the value of k.

m[2,4] = min(m[2,2] + m[3,4] + 5.89.34, m[2,3] + m[4,4] + 5.3.34)

m[2,4] = min(0 + 9078 + 15130, 1335 + 0 + 510) = min(24208,1845) = 1845

This minimum of m[2,4] is obtained using 3 as the value of k.

Now, inserting these values into the above table along with the corresponding k values, we arrive at:

1	0			
2	5785	0		
3	1530 (k=1)	1335	0	
4		1845 (k=3)	9078	0
	1	2	3	4

j

i

Finally, we calculate m[1,4] as,

m[1,4] = min(m[1,1] + m[2,4] + 13.5.34, m[1,2] + m[3,4] + 13.89.34, m[1,3] + m[4,4] + 13.3.34)

m[1,4] = min(0 + 1845 + 2210, 5785 + 9078 + 39338, 1530 + 0 + 1326)

m[1,4] = min(4055,54201,2856) = 2856

We get the minimum of m[1,4] when we use 3 as the value of k.

Inserting the value of m[1,4] along with the corresponding k value, we finally arrive at the following completed table:

j

1	0			
2	5785	0		
3	1530 (k=1)	1335	0	
4	2856 (k=3)	1845 (k=3)	9078	0
	1	2	3	4

i

Hence, the fewest number of multiplications needed to multiply the four matrices of the given dimensions is 2856.

To find out the order of multiplication, we just need to follow the value of k in the table.

The value of k for m[1,4] is 3 and hence the last multiplication that has to be done is the third one. i.e, (P.Q.R).S

Now, since the value used for k was 3, we need use the value of k at m[1,3] which is 1. This indicates that the first multiplication is the last but one multiplication to be done which leads to the second multiplication to be done first. Hence, the order of multiplication becomes (P.(Q.R)).S

Hence, ***the fewest number of multiplications required is 2856 and the order in which they have to be multiplied is generated as (P.(Q.R)).S.***

2.

No, the given method will not result in optimal solution always and the following is a counter example to prove the same.

Let us consider three matrices A, B and C with dimensions 12x13, 13x14 and 14x15 respectively.

According to the strategy given, the first multiplication has to be done between B and C since 14 is the largest remaining shared dimension. This results in the order of multiplication to be A.(B.C). This order requires 5070 multiplications to be processed.

Now, let us consider an alternate order of multiplication (A.B).C. This order requires 4704 multiplications which is much lesser than the previous case.

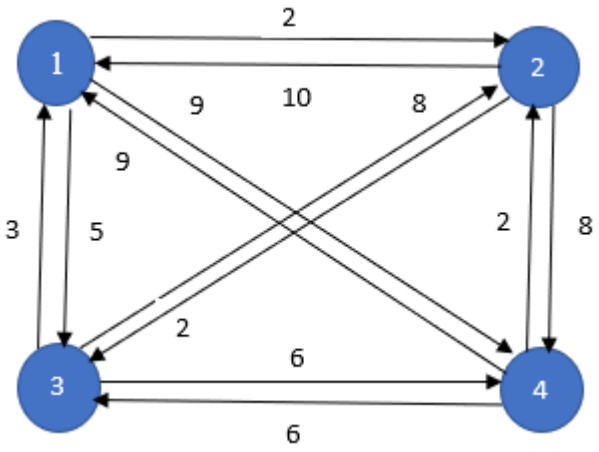
Hence, this acts as a counter example to prove that ***the strategy of using the largest remaining shared dimension for multiplication does not always minimizes the number of multiplications and does not always results in optimal or the best solution.***

3.

Given distance matrix:

$$M = \begin{bmatrix} 0 & 2 & 5 & 9 \\ 10 & 0 & 2 & 8 \\ 3 & 8 & 0 & 6 \\ 9 & 2 & 6 & 0 \end{bmatrix}$$

Converting the above adjacency matrix into graph format, we get the below graph:



Consider the set of nodes to be V = {1,2,3,4}

To find the optimal solution to the Travelling Salesman Problem for the given data set, we need to find the value of g(1,V-{1}) where g(i,S) is given as the length of the shortest path from the node i to node 1 passing through all the nodes in S exactly once.

Using dynamic programming, as discussed in the class, the value of g(i,S) can be calculated using the following formula:

$$g(i, S) = \min_{j \in S} (L_{ij} + g(j, S - \{j\}))$$

To find: g(1,V-{1}) = g(1,{2,3,4})

In order to find the value of g(1,{2,3,4}), the following calculations will be helpful:

$g(2,\phi) = L_{21} = 10$

$g(3,\phi) = L_{31} = 3$

$g(4,\phi) = L_{41} = 9$

$g(3,\{4\}) = L_{34} + g(4,\phi) = 6 + 9 = 15$

$g(4,\{3\}) = L_{43} + g(3,\phi) = 6 + 3 = 9$

$g(2,\{4\}) = L_{24} + g(4,\phi) = 8 + 9 = 17$

$g(4,\{2\}) = L_{42} + g(2,\phi) = 2 + 10 = 12$

$g(2,\{3\}) = L_{23} + g(3,\phi) = 2 + 3 = 5$

$g(3,\{2\}) = L_{32} + g(2,\phi) = 8 + 10 = 18$

$g(2,\{3,4\}) = \min(L_{23} + g(3,\{4\}), L_{24} + g(4,\{3\})) = \min(2 + 15, 8 + 9) = \min(17, 17) = 17$

$g(3,\{2,4\}) = \min(L_{32} + g(2,\{4\}), L_{34} + g(4,\{2\})) = \min(8 + 17, 6 + 12) = \min(25, 18) = 18$

$g(4,\{2,3\}) = \min(L_{42} + g(2,\{3\}), L_{43} + g(3,\{2\})) = \min(2 + 5, 6 + 18) = \min(7, 24) = 7$

Finally,

$g(1,\{2,3,4\}) = \min(L_{12} + g(2,\{3,4,\}), L_{13} + g(3,\{2,4\}), L_{14} + g(4,\{2,3\})$

$g(1,\{2,3,4\}) = \min(2 + 17, 5 + 18, 9 + 7) = \min(19, 23, 16) = 16$

Therefore, the optimal distance to the Travelling Salesman Problem is 16 units.

The minimum for g(1,{2,3,4}) comes when we travel to node 4 from node 1. Continuing the path, the minimum for g(4,{2,3}) comes when we travel from node 4 to node 2 and then finally, we go to node 3 and then back to 1.

Hence, ***the node ordering in the optimal tour is generated as 1 → 4 → 2 → 3 → 1 and the total distance that has to be covered in this optimal tour is 16 units.***

4.

Special Node Set (SNS): Given a graph G= (V,E), a subset of nodes S is called a special node set (also known as an independent set) if there are no edges between the nodes in S.

Largest Special Node Set (LSNS): The SNS of the largest cardinality is referred to as the LSNS (also known as the max independent set).

LSNS is a notoriously hard problem. We can solve a constrained case of this problem where G is a tree.

In a tree, there exists only two cases at a particular node:

1. Do not include node.

2. Include the node but do not include it's children.

The first step is to define the subtask.

LSNS(n) = size of the largest independent set subtree at node n.

We need to find LSNS(r) where r is the root of the tree.

The next step is to find out the Recursive Function or Recurrence relation and analyse it.

LSNS(j) = max { ΣLSNS(k) for each k ϵchildren(n), 1 + ΣLSNS(k) for each k ϵ grandchildren(n)}

This is what it says,

If node n is a member of SNS then LSNS = 1+ Sum of LSNS of all the grandchildren of n.

If node n is not a member of SNS then LSNS = Sum of LSNS of all children of node n.

Therefore LSNS(X) = max {1+ sum of LSNS of all grandchildren of X, sum of LSNS of all children of X}

Solving this problem recursively will lead to the calculation of the same subproblems again and again. Hence, we store the solutions of the subproblems using dynamic programming in order to avoid recalculating.

Dynamic Programming Algorithm:

Consider the function's name to be LargestSpecialNodeSet.

Steps:

```
initialize result_set = {}
if (node == None):
    return 0
```

```
if (node! = 0):
    return T[node]
child_node = {}
for edge in root.edges:
   child_node.add (edge.getAdjacentVertex())
grand_child_node = {}

for child in child_node:
   for edge in child.edges:
      grand_child_node.add(edge.getAdjacentVertex())

node_present_LSNS =1 //one for the corresponding node itself

for grand_child in grand_child_node:
 node_present_LSNS+=LargestSpecialNodeSet(grand_child,T,Set)

node_absent_LSNS = 0

for child in child_node:
    node_absent_LSNS+==LargestSpecialNodeSet(child,T,Set)

T[node] = max(node_present_LSNS, node_absent_LSNS)
If T[Node] == node_present_LSNS:
     result_set.add(node)
Return result_set
```

Time Complexity:

For the above dynamic programming approach, we get the time complexity as O(n).

The above algorithm does not work for a graph which is not a tree as this algorithm works only for binary trees. This algorithm cannot be used whenever the tree has 3 or more sub nodes to the root node.