

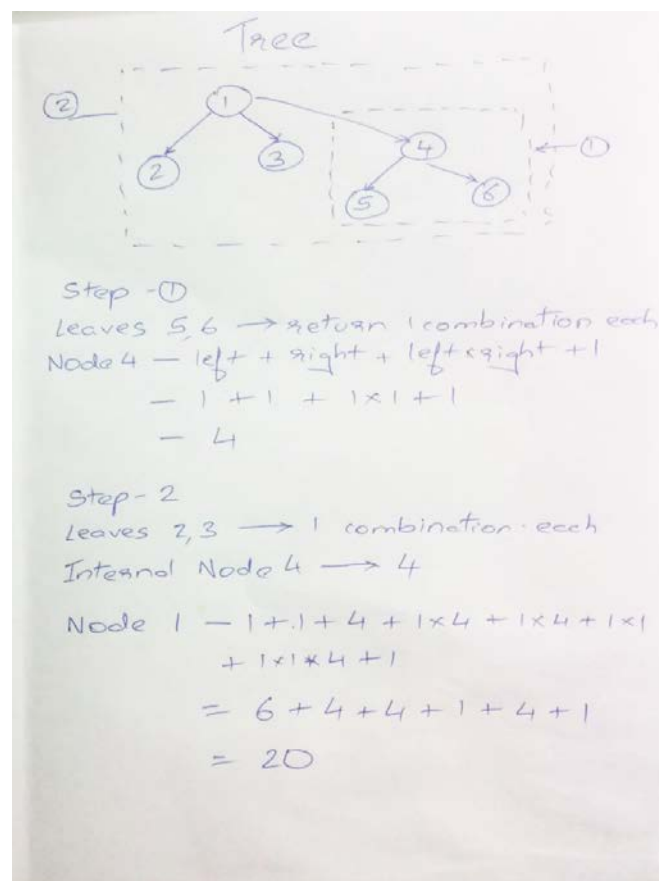
## Project Report

We have proposed two algorithms, one which efficiently enumerates consistent sub-DAGs in a tree while the other enumerates for a graph, too.

### Description

Algorithm 1(works for tree only):

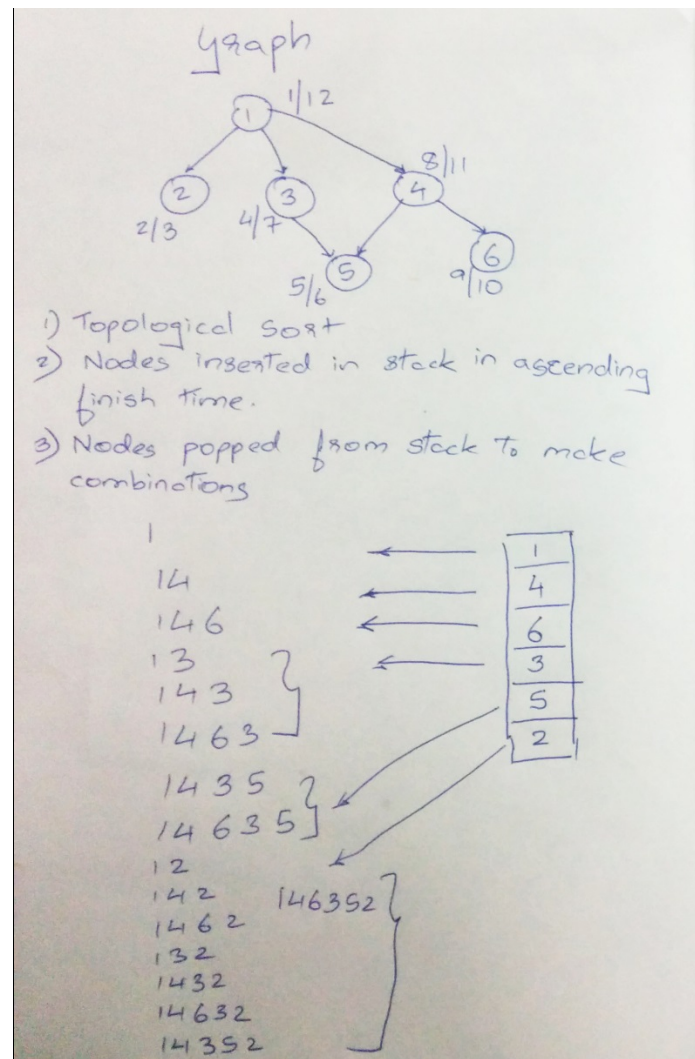
We follow a bottom up approach(DFS), that is, we start from the leaf node(u) and consider a sub-tree up to one level(which includes u, all its siblings and its parent v). For that sub-tree, we find the number of consistent sub-DAGs. This sub-tree(which would be represented by a count of consistent sub-DAGs) forms a left subtree or a right subtree for some parent and we keep calculating the consistent sub-DAGs for the above levels as shown in the example below:



Algorithm 2(works for trees and graphs as well):

We first call topologicalSorting() that traverses nodes in DFS and pushes them into a stack when finish time is assigned to a node. Next, we pop the nodes from the stack and check whether for each node (v) popped, all of its parent(s) from revadjacency list (that stores child->parent relations) is/are present or not in the set of consistent sub-DAGs generated. If

the parent(s) is/are absent, remove all those sub-DAGs from consideration. For all the other sub-DAGs left, add them into the list of consistent sub-DAGs by appending 'v' to them.



## Correctness

### Algorithm 1:

The DP approach maintains a count of consistent sub-DAG from left sub-tree and a count of consistent sub-DAGs from right sub-tree and a combination of consistent sub-DAG from left sub-tree and right sub-tree. Say sub-tree rooted at  $T$  has children  $v_i, v_{i+1}, \dots, v_j$ . Then, the total numbers of combinations for that sub-tree are:

for  $k=i$  to  $j-1$

$$T(v_k) = T(v_k) + N(v_k) + N(v_{k+1}) + N(v_k) * N(v_{k+1})$$

where  $N(v_k)$  represents the number of combination a  $v_k$  makes with its parent  $T$  and  $T(v_k)$  accumulates the count of all the combination amongst the children and parent. This technique is applied to all the sub-trees from the bottom to the top at all the levels.

## Algorithm 2:

We need to make sure that before we visit any node, all of its parent must be visited. This can be done by using a topological sort. Say for 2 nodes  $u, v \in V$ , if there is an edge from  $u$  to  $v$ , then the finish time for  $v$  will be less than  $u$  which means that node  $u$  (parent) was visited before node  $v$  (child). This algorithm will list consistent sub-DAGs because every new consistent sub-DAGs are created by scanning for its parent(s) in the set of preceded consistent sub-DAGs. For lookup, we have used KMP algorithm to reduce the computation time as much as possible.

## Complexity

### Analysis of Algorithm 1:

The algorithm follows a Depth-First Search which would take  $O(|V| + |E|)$  time. The cost of calculating the count of consistent sub-DAGs  $T(v_k)$  will be constant.

$$T(n) = O(|V| + |E|)$$

### Analysis of Algorithm 2:

Here the `topologicalSorting()` internally calls `dfsVisit()` that pushes a node by their finish time. This would take  $O(|V| + |E|)$ . Next, we pop every node  $v$  from stack, and scan the list of consistent sub-DAGs already. For each node  $v$ , pre-processing will take  $O(M)$  where  $m$  is length of node ' $v$ ' and the matcher will take  $O(C)$  where ' $C$ ' is a consistent sub-DAG previously formed. This search is done for  $2^{v-1}$  consistent sub-DAGs for every  $v$ .

$$T(n) = O(|V| + |E| + (v * M)(2^{v-1} * C))$$

## Counts of consistent sub-DAGs for the data-sets:

mini-mfo2: 692168

tree\_25: 89960

tree\_100: 185699666138602844715676

## Contributions

Algorithm 1 was proposed and implemented by Amish Shah.

Algorithm 2 was proposed and implemented by Janak Bhalla and Karan Shah.