

# **PROJECT REPORT**

## **ON**

# **CONSISTENT DAG'S**

(Submitted towards partial fulfillment of Course CSCI-B503)



**SCHOOL OF INFORMATICS  
AND COMPUTING**

---

**INDIANA UNIVERSITY**  
Bloomington

**UNDER THE GUIDANCE OF**

Predrag Radivojac

(Professor of Computer Science and Informatics, IUB)

**SUBMITTED BY**

Vikrant Kaushal

Shalabh

Ajay Saini

# ABSTRACT

This project work discusses design and analysis of an algorithm that enumerates all consistent sub-DAGs for a given Directed Acyclic Graph (DAG). We have assumed that the DAG has a single root node i.e. a node without any parents. A sub-DAG,  $H = (V_h, E_h)$  of  $G$  is called consistent if it satisfies the following three properties:

1.  $V_h \subseteq V_g$  and  $E_h \subseteq E_g$
2.  $\forall u, v \in V_h, (u, v) \in E_g \iff (u, v) \in E_h$
3.  $\forall v \in V_h \implies \text{Parents}(v, V_g) \subset V_h$

where  $\text{Parents}(v, V_g)$  defines a set of nodes  $u \in V_g$  such that  $(u, v) \in E_g$ . In other words, given a DAG  $G$ , the consistent sub-DAG is constructed by selecting a subset of nodes from  $V_g$  (such that the “parents” property holds) and keeping all edges from  $E_g$  that start from and end in the nodes from  $V_h$ .

The challenge was to design an algorithm that must be sub-exponential in the number of nodes  $V_g$ ; that is, it is not allowed to generate all subsets of the set of nodes  $V_g$  and then check if each such subset is consistent.

We have designed the algorithm for two different cases:

**Case 1:** When the DAG is a tree i.e. each node can have at most one parent, but it can have any number of children.

**Case 2:** When the DAG is not a tree i.e. each node can have any number of parents and any number of children.

### **Case 1: DAG is a tree**

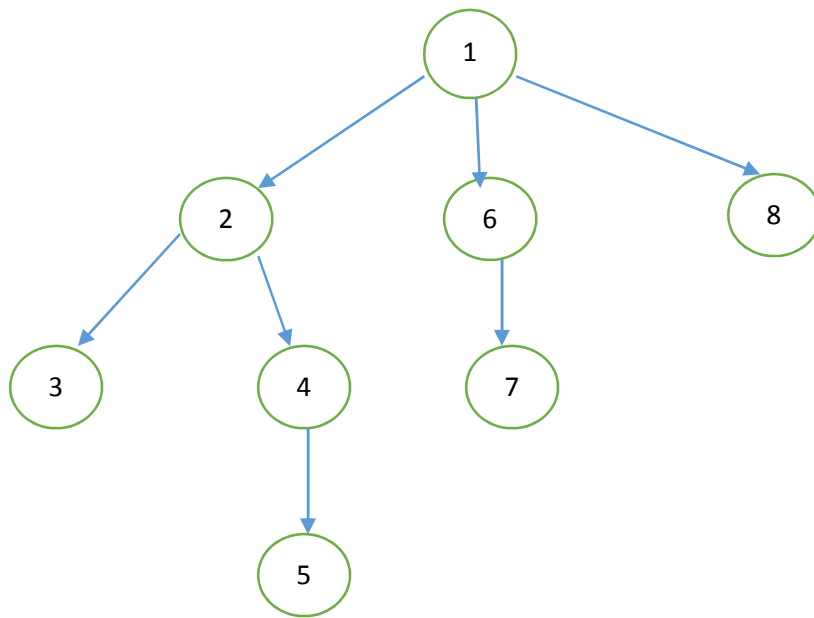
To count the number of all consistent sub-DAG's, we will use Depth First Search (DFS) method i.e. first of all we will process leftmost subtree completely and then move to right subtrees. We have divided the tree into sub-trees by assuming the root node is absent and its children are now the roots of different trees, which are referred to as nodes at level 1. We will add the original root node at the end of calculation because it will be adding only 1 consistent DAG. For each node we will store the following values: Value of Node = Value of parent; Value of parent =  $2 * \text{Value of parent}$ ; Value of all grandparents = Value of itself + Value of node. The total number of DAG's are calculated using the formula for the first two nodes (Value of node 1) + (Value of node 1 \* Value of node 2) + (Value of node 2) and for the subsequent nodes (Result of previous nodes) + (Result of previous nodes \* Value of node) + (Value of node). Finally 1 is added to the result corresponding to the original root node.

#### **Algorithm:**

##### **CONSISTENT\_TREE()**

1. Extract and store the root of the tree.
2. Convert the given matrix representing the nodes of tree into a HashMap depicting the child – parent relationship among the nodes.
3. Traverse the tree in DFS manner.
4. for all the nodes at level 1
5.       set value = 1
6. for i = 2 to n
7.       Value of node = Value of parent
8.       Value of parent =  $2 * \text{Value of parent}$
9.       Value of all grandparents = Value of itself + Value of node
10. for all the nodes at level 1,
11.     take the first 2 nodes from left
12.       Tot\_DAG = (Value of node 1) + (Value of node 1 \* Value of node 2) + (Value of node 2)
13. do following from j= 3 until rightmost node
14.       Tot\_DAG = (Tot\_DAG) + (Tot\_DAG \* Value of node j) + (Value of node j)
15. Total number of consistent DAG's = Tot\_DAG + 1

## EXPERIMENTAL ANALYSIS:



Node 2 = 1	Node 6 = 1	Node 8 = 1
Node 3 = 1	Node 7 = 1	
Node 2 = 2	Node 6 = 2	
Node 4 = 2		
Node 2 = 4		
Node 5 = 2		
Node 4 = 4		
Node 2 = 6		

According to our formula to calculate total no. of DAG's:-

Step 1: Tot\_DAG = (Value of node 2) + (Value of node 2 \* Value of node 6) + (Value of node 6)


$$= 6 + 6 * 2 + 2$$

$$= 20$$

Step 2: Tot\_DAG = Tot\_DAG + Tot\_DAG \* Value of node 8 + Value of node 8

$$= 20 + 20 * 1 + 1$$

$$= 41$$

Step 3: Total no. of DAG's = Tot\_DAG + 1  (for root node)

$$= 41 + 1$$

$$= 42$$

Manually counting the consistent DAG's:

1. {1}
2. {1,2}
3. {1,2,3}
4. {1,2,4}
5. {1,2,3,4}
6. {1,2,4,5}
7. {1,2,3,4,5}
8. {1,6}
9. {1,2,6}
10. {1,2,3,6}
11. {1,2,4,6}
12. {1,2,4,5,6}
13. {1,2,3,4,6}
14. {1,2,3,4,5,6}
15. {1,6,7}
16. {1,2,6,7}
17. {1,2,3,6,7}
18. {1,2,4,6,7}
19. {1,2,4,5,6,7}
20. {1,2,3,4,6,7}
21. {1,2,3,4,5,6,7}
22. {1,8}
23. {1,2,8}
24. {1,2,3,8}

- 25. {1,2,4,8}
- 26. {1,2,4,5,8}
- 27. {1,2,3,4,8}
- 28. {1,2,3,4,5,8}
- 29. {1,2,6,8}
- 30. {1,2,3,6,8}
- 31. {1,2,3,4,6,8}
- 32. {1,2,4,6,8}
- 33. {1,2,4,5,6,8}
- 34. {1,2,3,4,5,6,8}
- 35. {1,2,6,7,8}
- 36. {1,2,3,6,7,8}
- 37. {1,2,4,6,7,8}
- 38. {1,2,4,5,6,7,8}
- 39. {1,2,3,4,6,7,8}
- 40. {1,2,3,4,5,6,7,8}
- 41. {1,6,8}
- 42. {1,6,7,8}

## **Case 2: DAG is not a tree**

When DAG is not a tree, then we process the elements of graph in Breadth First Search (BFS) manner i.e. we cannot move to next level until we are finished processing all the elements at the current level. Again, we will divide the tree into sub-trees by assuming the root node is absent and its children are now the roots of different trees. We will add the original root node at the end of calculation because it will be adding only 1 consistent DAG.

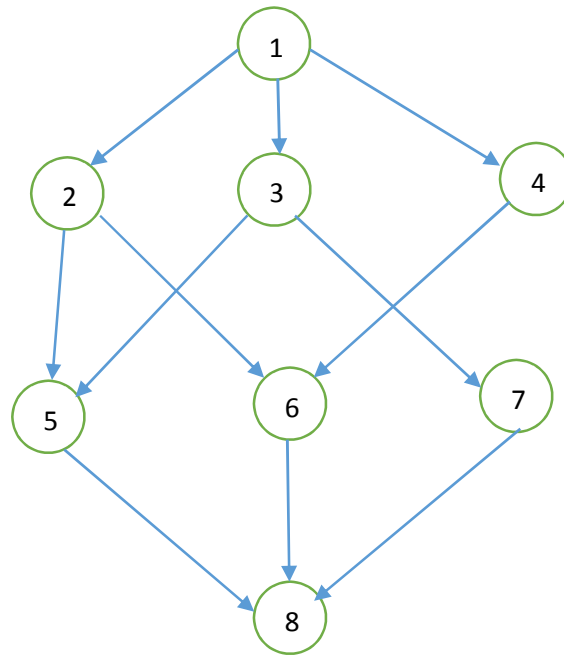
First of all, we will read the data in the way it is provided. Then starting from the leftmost node at level 1 and moving subsequently in the right direction, we will store the {node, node – ancestor} relation in a HashMap, say HM1, for each node. Simultaneously, we will store all the consistent DAG's in another HashMap, say HM2. When we encounter a new node, for it to be consistent we have to track down all of its grandparents. Since, we are traversing in Breadth First Search manner, before discovering a node all of its parents and grandparents would have been already discovered. So, we will use **Dynamic Programming** concept here i.e. for each new node, we will use the relations of its parents and grandparents stored in HM1. Through this way, we have also ensured that there will be no duplicate copies of consistent DAG's.

### **Algorithm:**

CONSISTENT\_DAG:

1. Extract and store the root of the graph.
2. Randomly read the data as provided.
3. Iterate the graph in breadth first search (BFS) manner.
4. for each node
5.     store {node, node – ancestor} relation in a HashMap.
6.     Using dynamic programming store all the consistent DAG's in another HashMap.

## **EXPERIMENTAL ANALYSIS:**



### **HASH-MAP 1:**

1. {2, 2-1}
2. {3, 3-1}
3. {4, 4-1}
4. {5, 5-2, 5-3, 2-1, 3-1}
5. {6, 6-2, 6-4, 2-1, 4-1}
6. {7, 7-3, 7-4, 3-1, 4-1}
7. {8, 8-5, 8-6, 8-7, 5-2, 5-3, 2-1, 3-1, 6-2, 6-4, 4-1, 7-3, 7-4}

### **HASH-MAP 2 (Consistent DAG's):**

1. {1}
2. {2-1}
3. {3-1}
4. {3-1, 2-1}
5. {4-1}
6. {4-1, 2-1}
7. {4-1, 3-1}
8. {4-1, 3-1, 2-1}
9. {5-2, 2-1, 5-3, 3-1}
10. {5-2, 2-1, 5-3, 3-1, 4-1}



11. {6-2, 6-4, 2-1, 4-1}
12. {6-2, 6-4, 2-1, 4-1, 3-1}
13. {6-2, 6-4, 2-1, 4-1, 3-1, 5-2, 5-3}
14. {7-3, 7-4, 3-1, 4-1}
15. {7-3, 7-4, 3-1, 4-1, 2-1}
16. {7-3, 7-4, 3-1, 4-1, 2-1, 5-2, 5-3}
17. {7-3, 7-4, 3-1, 4-1, 2-1, 6-2, 6-4}
18. {7-3, 7-4, 3-1, 4-1, 2-1, 5-2, 5-3, 6-2, 6-4}
19. {8-5, 8-6, 8-7, 5-2, 5-3, 2-1, 3-1, 6-2, 6-4, 4-1, 7-3, 7-4}

## **REFERENCES**

1. Introduction to Algorithms by Thomas H. Cormen

## **Contributions of Each Member**

Problem Formulation – Vikrant, Shalabh, Ajay

Coding – Vikrant, Shalabh, Ajay

Report - Vikrant, Shalabh, Ajay