Enumerating Consistent Sub-DAG's in a DAG

Shawn Peng

December 11, 2015

At the beginning, I want to define some symbols for convenience.

Define function $g : \{G \mid G \text{ is a DAG } \} \to \mathbb{N}$, for any (DAG) G, g(G) gives the number of consistent sub-DAG's of G.

For any $v \in V_G$, we can also use v to represent the sub-DAG which contains all vertices and edges that can be achieved from v.

Then, we can also define function g on vertices. For $v \in V_G$, $g(v)$ gives out the number of consistent sub-DAG's of the sub-DAG v.

Define minus operation on sub-DAG's of G. For $u, v \in V_G$, and v can be achieved from u, then $u - v$ gives the sub-DAG consisting of all vertices can be reached from u but can't be reached from v. In other words, take sub-DAG u and then remove sub-DAG v.
Now we can deal with the tree problem.

Problem a:

The DAG G is a tree.
Assume the root of G is a, then

$$g(G) = g(a) = \prod_{(a,v) \in E_G} (1 + g(v)) \tag{1}$$

Proof:
For any vertex u of G, and a child of u, v, any consistent sub-DAG of u is either containing v or not, and for any consistent sub-DAG that not containing v, we can combine it with any consistent sub-DAG of v and it's still a consistent sub-DAG of u. So for any consistent sub-DAG of u not containing v, there are $g(v)$ different possibilities that containing v.

The number of consistent sub-DAG that not containing v is equal to the number of consistent sub-DAG $u - v$, because since v is not represent, any vertex in v can't represent in a consistent sub-DAG, and then it doesn't matter if v is there.
That is

$$\begin{aligned} g(u) &= g(u - v) + g(u - v)g(v) \\ &= g(u - v)(1 + g(v)) \end{aligned}$$

Then for any child x of u, we can remove the sub-DAG x from u and then multiply $g(x)$. When we removed all children, the number of consistent sub-DAG's of a single vertex DAG is equal to one. So that, equation 1 holds.

Algorithm:

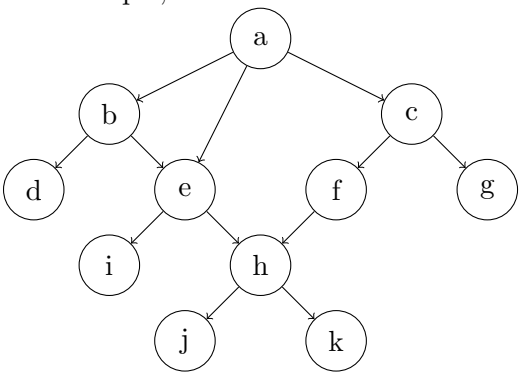
function $g_T(u)$
  if u has no child **then**
    return 1
  end if
  c = 1
  for v in children of u **do**
    $x \leftarrow g_T(v)$
    $c \leftarrow c * (1 + x)$
  end for
  return c
end function

The time complexity is $O(n)$ for linking-list representation. This is related to the complexity of finding a vertex and finding children of a vertex.

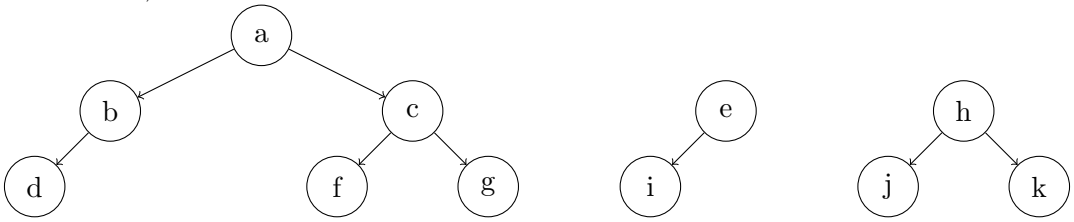
Problem b:

DAG G is not a tree. (We always assume G only have one root.) Then the difficulty is that for those vertices that have multiple parents, we cannot combine them directly with the possibilities that don't contain them. To solve this problem, we can first remove all sub-DAG's root at vertices which have multiple parents.

For example, if we have a DAG G:



which can become,



Then we can easily calculate the numbers $g_T(a - e - h)$, $g_T(e - h)$, and $g_T(h)$. (To distinguish the function that can only apply on a tree, we use the subscription T to denote it afterward.) Now the problem is that how can we get $g(a)$

To do this, we introduce a new symbol '|' for function g.

$$\forall u \in V_G, g(G|u) \equiv \text{the number of consistent sub-DAG's of } G \text{ that containing vertex } u$$

We can read $g(G|u)$ as "g of G given u", similar to what we have in probability.

We can prove that,

$$g(G) = g(G - u) + g(G|u) \tag{2}$$

where G doesn't need to be a tree.

Proof:
For any vertex u in G, every consistent sub-DAG of G must either containing u or not, and the number of those that containing u is $g(G|u)$ and the number of those not containing u is equal to $g(G - u)$. It can't be both or neither, so that the equation 2 holds.

If r is root of G, and u is vertex that have only one parent, and u is a independent sub-DAG, then

$$g(r|u) = g(r - u)g(u) \tag{3}$$

Define: a sub-DAG of G is independent if and only if all vertices in it doesn't have any parent outside this sub-DAG, except the root of the sub-DAG.

Proof:
Because u and $r - u$ are independent, this is equivalent to a tree problem. For any consistent sub-DAG that not containing u, we can combine it with any consistent sub-DAG of u and it's still a consistent sub-DAG of r. That is,

$$g(u) = g(r - u)(1 + g(u))$$

because we also have

$$g(u) = g(r - u) + g(r|u)$$

so that,

$$g(r|u) = g(r - u)g(u)$$

When u have multiple parents, we need to consider those vertices that u depending on. Not all possible of consistent sub-DAG's of $r - u$ can combine with possibilities in u. So the possibilities will be less than before.

First we have that for any $u \in V_G$, sub-DAG u is independent,

$$g(r|u) = g(r|D(u)) \tag{4}$$

Define function $D(u)$, where u is a vertex of G, gives the sub-DAG of G that the vertex u is depending on. It is the smallest consistent sub-DAG that containing u.

Proof:
According to the definition of consistent sub-DAG, any consistent sub-DAG containing u will need to containing all vertices and edges in $D(u)$, that's also why there exists a smallest consistent sub-DAG containing u.

Then for $g(r|D(u))$, if we assume for all $v \in V_{D(u)}$, $v - D(u)$ is independent, then we have that,

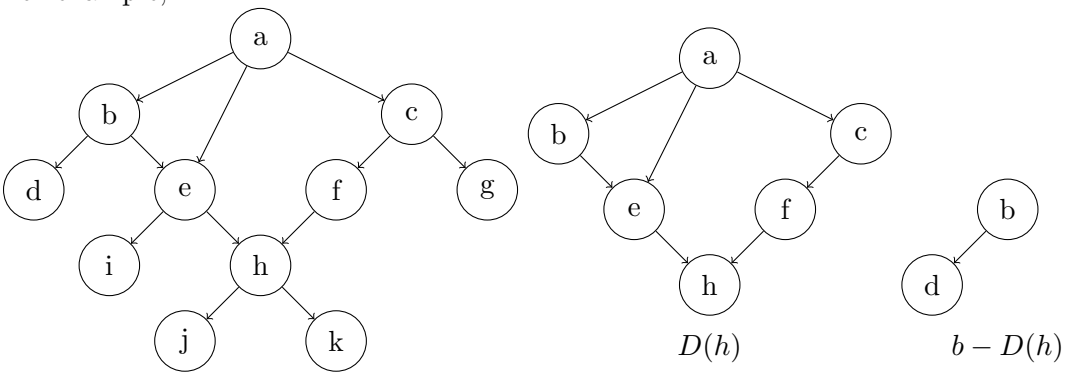
$$g(r|D(u)) = \prod_{v \in D(u)} g(v - D(u)) \tag{5}$$

(Note: u and r are also in $D(u)$)

Proof:

First, we need to redefine the minus operation:
$\forall v \in V_G$, S is a sub-DAG of G, $v - S \equiv$ the sub-DAG is generated by removing all vertices from sub-DAG v that is in S except the root of sub-DAG v which is v itself. So $u - D(u) = u$, there is nothing to remove.

For example,



$D(h)$ $\quad$ $b - D(h)$

Because $v - D(u)$ is independent so that we can combine any consistent sub-DAG of $v - D(u)$ with any other consistent sub-DAG of w, $w \in V_{D(u)}$. So that's a Cartesian product of sets of consistent sub-DAGs of $w \in V_{D(u)}$, and any sub-DAG in this product is unique consistent sub-DAG. Also those are all the possibilities that $g(r|D(u))$ can have. We can see this in another way. First, we take an empty set C. Then, similar to the root case, we take $D(u)$ as the first case, which is a consistent sub-DAG itself, and we add it to the set C. Then we iterate all the vertices in $V_{D(u)}$, and for any vertex v, we combine all the possibilities of consistent sub-DAG's of $v - D(u)$ with any possibility in C, except the consistent sub-DAG that only contains of vertex v, which has already in C, and add them into C. So that, for i'th iteration,

$$\begin{aligned} |C^{(i)}| &= |C^{(i-1)}|[1 + g(v_i - D(u)) - 1] \\ &= |C^{(i-1)}|g(v_i - D(u)) \end{aligned}$$

Initial condition is $|C^{(0)}| = 1$, so when the iteration ends, we get

$$|C| = g(r|D(u)) = \prod_{v \in D(u)} g(v - D(u))$$

Now, the problem can be solved, if there doesn't exist two vertex $u, v \in V_G$ such that u, v both have more than one parent and also are independent.

First, decompose the DAG into trees from the vertices that have multiple parents. Then we calculate $g(u)$ for every vertex u, and store it into the vertex's data. Cause for any u, we only need $g(x)$ for all its children, all this process can be done in linear time for adjacent list representation.

Then we try to put back all these trees. When we put them back, we need to make sure that all parents are existing in current DAG G'. If not, we can just move it to the end of the list. For each tree with root u, we need to get $D(u)$ and then use equation (5).

When we put back these trees, there may appear two possibilities. One is that we never find some vertex with multiple parents exists in the current DAG G' but not in $D(u)$. The second one is that we find some. Now we try to deal with the first case.

To calculate $g(v_i - D(u))$, we can use the values that we have, and we actually need only to check whether v_i's children are in $D(u)$. There are several cases,

$\forall w, (v_i, w) \in E_G$
1. $w \notin V_{D(u)}$, we can skip this vertex
2. $w \in V_{D(u)}$,

a. w has only one parent, we need to divide $g_T(v_i)$ by $1 + g_T(w)$, because if w has only one parent, it must be in the tree v_i when we calculate $g_T(v_i)$, so that,

$$g_T(v_i - w) = \frac{g_T(v_i)}{1 + g_T(w)} \tag{6}$$

b. w has multiple parent, we do need to do the division. That's because w can't be in the tree. Therefore $g_T(v_i - w) = g_T(v_i)$

To sum up, Define $C(u)$ gives the children vertices of u, $P(u)$ gives the parents of u.

$$g_T(v_i - D(u)) = \frac{g_T(v_i)}{\prod_{w \in C(v_i), w \in D(u), |P(u)|=1}(1 + g_T(w))} \tag{7}$$

For the second case, we can find all vertices that have multiple parents, and then take combination with these vertices and u. Consider a special case, say there just exists one such node w, then,

$$g(r) = g(r - u - w) + g(r - w|u) + g(r - u|w) + g(r|u, w) \tag{8}$$

That is take combination of cases containing u and not and cases of containing w or not.

Proof:
That is all possibilities and there isn't anyone appeared twice or more. For any consistent sub-DAG of G, it must be in exactly one case.

Obviously, w can't be in $V_{D(u)}$. If u is in $V_{D(w)}$, then $g(r - u|w) = 0$ and we don't need to compute it, and also w can't be in G' when we adding u, so that u can't be in $V_{D(w)}$, neither.

Now we are trying to adding u, we have already calculated $g(r-u-w), g(r-u|w)$, what we need are $g(r-w|u), g(r|u,w)$. For $g(r|u,w)$, we need to take a union of $D(u)$ and $D(w)$, let's denote as $D(u) \cup D(w)$.

Proof:
At now, u and w in G' should be independent, because any tree that depends on u can't be added before u. Therefore, similar as before, we can take the Cartesian product of sets of consistent sub-DAG's of all vertex $z \in V_{D(u)\cup D(w)}$

$$g(r|u,w) = g(r|D(u),D(w)) = \prod_{z \in V_{D(u)\cup D(w)}} g(z - D(u) - D(w)) \tag{9}$$

For $g(r-w|u)$, at first look that it seems we need to remove w from G'. Actually, from the case analysis we have done before, we can figure out that we never used the information whether some tree has been added to G'. Therefore, we can add them altogether to G' after we calculate $g_T(u)$ for all vertices. And

$$\begin{aligned} g(r-w|u) &= g(r-w|D(u)) \\ &= \prod_{z \in V_{D(u)}} g(z - D(u) - w) \\ &= \prod_{z \in V_{D(u)}} g_T(z - D(u)) \end{aligned}$$

All we need is the information that what are the vertices with multiple parents have been add to G', such that we can check whether there exists w that is independent to u.

After done with combination of two independent multi-parent vertices, we can generalize this method to apply to multiple number of such vertices that are independent to u. One special case is that all these vertices are independent to each other, then we can directly expanding the possibilities. If there are d such independent vertices, we will have 2^d possibilities, so if there are a lot number of such vertices, this will

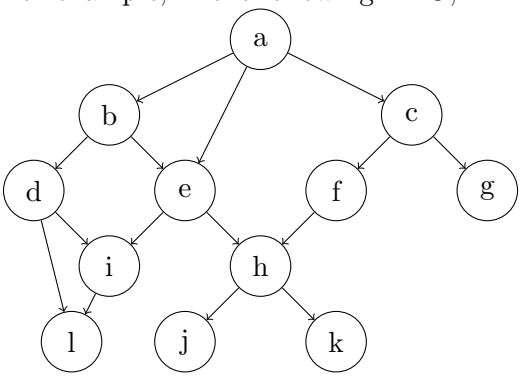
For general case, that in these vertices although they are independent to u, but maybe they keep some dependency between each other, we need to check the circumstance when we generate a possible path.

Given an multi-parent vertex w which is independent to the u which is the root of a sub-DAG that we are trying to add, we list all the cases.

1. If w is not in the path now, we may choose w or not.
    a. If we not choose w, we don't need to worry about anything. We just keep going to take possibilities of the next independent multi-parent vertex.
    b. If we choose w, we need to make sure that there isn't any vertex that w depending on is at not chosen status before we consider w, otherwise it will be a contradiction.

2. If w is in the path now, then we must choose it, and cause it's already in the path, we don't need to add $D(w)$ to the path again.

For example, in the following DAG,



Suppose our combination order is e, i, l, h. Now we try to add sub-DAG h, and we find there are two vertices independent to h, i, l. When we selecting l, we need to make sure we didn't leave out i before. If it is the case, i was not chosen before h, then we can only add the possibilities that not chosen h.

Now we can solve the problem in $O(n + k2^d d \log(n) \log(n))$ time, where k is the number of vertices that have multiple parents, and d is the maximum number of multi-parent vertices that are independent to each other. $O(n)$ for the calculation of $g_T(u)$, and for each multi-parent vertex, the worst case need to calculate 2^d different path. Every such path need $O(d \log(n))$ time, because for every multi-parent vertex, we need to get all its parents and parents of parents to the root, the depth is $O(\log(n))$ (we assume that the in-degree (fan-in factor) is a constant) and there are d such vertex we need to deal with. At last, for each path, we need $O(\log(n))$ time to calculate $g(u|Path)$.

This algorithm can solve medium scale problems where d is less than 20.

Results:

Tree 25	89960
Tree 100	185699666138602854678528
mini-mfo	2752806873600
mini-mfo2	692168