On Computing the Number of Topological Orderings of a Directed Acyclic Graph

Wing-Ning Li, Zhichun Xiao, Gordon Beavers
Department of Computer Science and Computer Engineering
University of Arkansas
wingning@uark.edu, zxiao@uark.edu, gordonb@uark.edu

Abstract

When can the number of topological orderings of a Directed Acyclic Graph (DAG) be efficiently determined? We propose a divide-and-conquer method that partitions a DAG into sub-graphs from which the number of topological orderings is calculated using combinatorial methods. The concepts of static vertex and static vertex set are introduced and are used to partition a DAG into sub-graphs. Algorithms are proposed to identify static vertices and static vertex sets, and their induced sub-graphs called static sub-graphs. Transitive closure is used in identifying the static vertex sets. Special structured DAGs that allow our method to achieve complete partitioning and open issues, such as sub-digraphs for which no obvious partitions can be found, are discussed. Preliminary experiments are conducted to study the effectiveness of the proposed algorithm in counting the number of topological orders.

Keywords: Directed acyclic graph, topological order, transitive closure, series-parallel digraph

1 Introduction

Directed acyclic graphs (DAG) are used to indicate a precedence relation (a partial order) among the modeled elements. Given a DAG $G = (V, E)$, a topological ordering of G is a linear (total) order of all the vertices which respects the partial order, that is, if G contains an edge (u, v), i.e., $u \prec v$, then u appears before v in the ordering. Figure 1 depicts a DAG and Figure 2 lists all the topological orders of the DAG. As can be seen from this example, generally many topological orders exist for a given DAG, that is, many total orders are consistent with a given partial order.

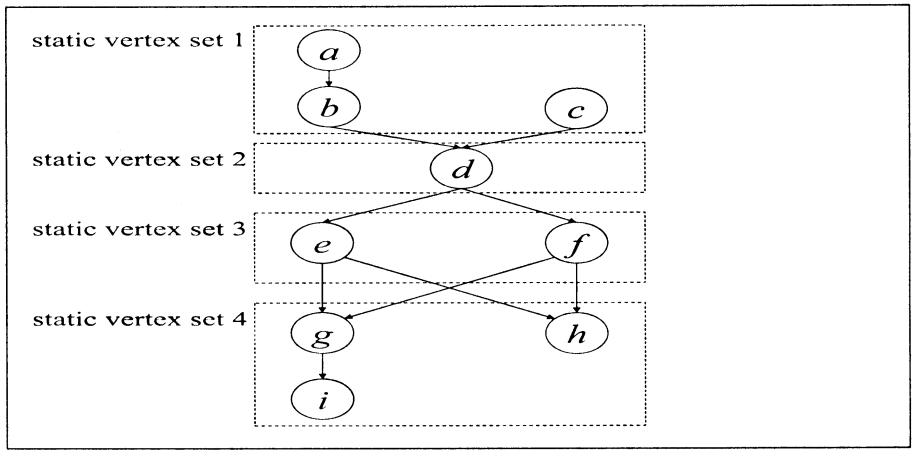

Figure 1: An example DAG and its static vertex sets.

c	a	b	d	e	f	g	h	i
a	c	b	d	e	f	g	h	i
a	b	c	d	e	f	g	h	i
c	a	b	d	f	e	g	h	i
a	c	b	d	f	e	g	h	i
a	b	c	d	f	e	g	h	i
c	a	b	d	e	f	h	g	i
a	c	b	d	e	f	h	g	i
a	b	c	d	e	f	h	g	i
c	a	b	d	f	e	h	g	i
a	c	b	d	f	e	h	g	i
a	b	c	d	f	e	h	g	i
c	a	b	d	e	f	g	i	h
a	c	b	d	e	f	g	i	h
a	b	c	d	e	f	g	i	h
c	a	b	d	f	e	g	i	h
a	c	b	d	f	e	g	i	h
a	b	c	d	f	e	g	i	h

Figure 2: All topological orders of the DAG in Figure 1.

For a given DAG, $G = (V, E)$, a topological order can be computed in many ways. For example, the set of in-degree zero vertices, V', is computed and output in any order. Then the same step is repeated recursively on the subgraph of G induced by $V \setminus V'$. The process is terminated when either the input graph is empty or the input graph is not empty but the set of in-degree zero vertices is. The first termination condition is reached when the input is a DAG and a topological order is produced. The second condition is reached when the input digraph has a cycle and no topological order can be produced. The time complexity of a literal implementation of the above algorithm is $O(n(n+e))$, where $n = \mid V \mid$ and $e = \mid E \mid$. However, the time complexity of a clever implementation of the algorithm, that uses a queue data structure and incrementally derives the induced subgraph and its in-degree zero vertices, is $O(n+e)$. The technique used here amounts to the breadth first search method. The depth first search method may also be used to develop another algorithm that produces a topological order with similar time complexity [1].

A related problem is to enumerate all possible topological orders of a given DAG. In other words, the interest one has is not in a single topological order but all such orders. The same method which produces one topological order may naturally be extended to a backtracking algorithm, which generates all possible topological orders. Notice that any vertex of V', the set of in-degree zero vertices, can take the first position in a topological order. And together, they create the choices for backtracking. In the backtracking scheme, each topological order is generated in $O(n + e)$ time so that the overall time is $O(K(n + e))$ where K is the number of topological orders of the given DAG. The reader is referred to [4, 2] for a more in depth discussion of algorithms and ideas on generating all topological sorts.

The dimension of a partial order P is the minimum number of linear orders (topological orders) whose intersection is P. The transitive closure of the edge set E of a DAG forms a partial order E'. So generating all topological orders is related to finding a minimum subset of whose intersection is E'. The linear time recognition algorithm [3] of series parallel digraphs uses the idea of dimensionality of partial order.

Another related problem this paper addresses is to determine the number of topological orders, K, of a given DAG. One approach to obtain such a number is to use the aforementioned method of total enumeration. Since computing K looks simpler than generating all topological orders, it might be solvable by a more efficient method. Indeed, a more efficient method is proposed. The method is a divide-and-conquer method that partitions a DAG into sub-graphs from which the number of topological orderings is calculated using combinatorial methods. The concepts of static vertex and static vertex set are used to partition a DAG into sub-graphs. The reader is referred to [5] for applications that motivates the study of this

problem. The related results and more thorough discussion can be found in the thesis.

The next section introduces the concepts of static vertex and static vertex set, and characterizes the properties of static vertex and static vertex set. Results of section 2 are used by other sections. Section 3 first gives a high level description of the divide and conquer algorithm and proves its correctness. Then algorithmic details of how to compute all static vertex sets of a given graph efficiently are provided. Section 4 considers dags with special properties that may further improve the efficiency of the algorithms. Section 5 presents preliminary test results. The last section concludes the paper with some open issues.

2 Preliminary

Let $G = (V, E)$ be a DAG. We say the DAG is *connected* if it is weakly connected, that is, when we disregard the arc directions, the graph is connected, i.e., has one component. A DAG is *disconnected* if it has more than one weakly connected component. A weakly connected component is referred to as a *component* in this paper.

An edge from u to v is denoted by $u \prec v$. A path from u to v is denoted by $u \prec^+ v$. $PRED(v)$ denotes the set of predecessors of v. Formally, we have $PRED(v) = \{u \mid u \prec^+ v\}$. $SUCC(v)$ denotes the set of successors of v. Formally, we have $SUCC(v) = \{u \mid v \prec^+ u\}$. Let $S \subset V$. We have $PRED(S) = \cap_{v \in S} PRED(v)$ and $SUCC(S) = \cap_{v \in S} SUCC(v)$. In plain English, $PRED(S)$ is the greatest common predecessor set in S and $SUCC(S)$ is the greatest common successor set. For the example DAG of Figure 1, we have $PRED(d) = \{a, b, c\}$, $SUCC(d) = \{e, f, g, h, i\}$, $PRED(f) = \{a, b, c, d\}$, $SUCC(f) = \{g, h, i\}$, $PRED(d, f) = \{a, b, c\}$, and $SUCC(d, f) = \{g, h, i\}$.

Now we formally introduce the concepts of *static vertex* and *static vertex set.*

Definition 1 *Vertex v is a* ***static vertex*** *if and only if* $|PRED(v)| + |SUCC(v)| = |V| - 1$.

A vertex set S is called trivial if $S = \emptyset$ or $S = V$, otherwise S is called nontrivial. The static vertex sets are nontrivial and are defined as follows.

Definition 2 *A nontrivial vertex set S is a* ***static vertex set*** *if and only if* $|PRED(S)| + |SUCC(S)| = |V| - |S|$ *and no proper subset of S has the same property.*

Definition 3 *The sub-graph induced by a static vertex set is called a* ***static sub-graph***

Based on the above definitions, it is straight forward to verify that for the DAG of Figure 1, d is a *static vertex*, f is not a *static vertex*, and $\{d, f\}$ is not a *static vertex set*. The *static vertex sets* of the DAG are $\{a, b, c\}$, $\{d\}$, $\{e, f\}$, and $\{g, h, i\}$. They are shown inside the boxes in Figure 1. Notice that a *static vertex* is a special case of a *static vertex set* where the set is a singleton. Notice also that $\{a, b, c, d\}$ is not a *static vertex set* even if the condition $\mid PRED(S) \mid + \mid SUCC(S) \mid = \mid V \mid - \mid S \mid$ is satisfied. It is not a *static vertex set* because it has a proper subset $\{d\}$ that is a *static vertex set.*

Let n be the number of vertices of the DAG G. A topological order may be viewed as an assignment of n vertices into n positions from 1 to n where u occupies an earlier position than that of v if $u \prec v$. A value between 1 to n is referred to as a *position.* A sequence of contiguous positions is referred to as a *sequence.* For example, *sequence*$[2, 4]$ presents positions 2,3, and 4.

Lemma 1 *A static vertex, v, can only appear at $\mid PRED(v) \mid +1$ position among all topological orders.*

Proof: Suppose v appears at position i, $i <\mid PRED(v) \mid +1$, in a topological order. Then only $i-1$ positions are available before i. Since $i <\mid PRED(v) \mid +1$, we have $i - 1 <\mid PRED(v) \mid$. This implies that at least one of the vertices in $PRED(v)$ must appear after v, a contradiction of the definition of topological order. Since $|PRED(v)| + 1 = |V| - |SUCC(v)|$, for a static vertex, a similar contradiction will be derived when $i > |V| - |SUCC(v)|$. □

Lemma 2 *Each member of a static vertex set, S, can only appear in the sequence$[|PRED(S)| + 1, |V| - |SUCC(S)|]$ in any topological order.*

Proof: Since $PRED(S)$ is the greatest common predecessor set of vertices in S and $SUCC(S)$ is the greatest common successor set, the Lemma follows from the same arguments used in Lemma 1 □

Dotted vertical lines of Figure 2 illustrate both lemma 1 and lemma 2.

Lemma 3 *Let S_1 and S_2 be two different static vertex sets. $S_1 \cap S_2 = \emptyset$.*

Proof: Suppose $S_1 \cap S_2 = A \neq \emptyset$. Since S_1 and S_2 are different, $A \subset S_1$. Then the fact that $\mid PRED(A) \mid + \mid SUCC(A) \mid = \mid V \mid - \mid A \mid$ contradicts that S_1 is a static vertex set because the definition of a static vertex set does not allow its proper subset to have this property. $\mid PRED(A) \mid + \mid SUCC(A) \mid = \mid V \mid - \mid A \mid$ follows from the observation that vertices in $S_1 \setminus S_2$ and $S_2 \setminus S_1$ are partitioned into $PRED(A)$ and $SUCC(A)$ due to $A \subset S_2$ and S_2 being static, and $A \subset S_1$ and S_1 being static respectively. □

Corollary 1 *If a graph G has a static vertex set that is a proper subset of the vertices of G, then G can be partitioned into static sub-graphs.*

The next corollary follows from Lemmas 2 and 3.

Corollary 2 *If the vertex set $V(G)$ of a dag G is composed of k static vertex set $S_1, \cdots, S_k$, then the sequence $[|PRED(S_i)|+1, |V|-|SUCC(S_i)|]$, $1 \leq i \leq k$, are disjoint.*

Given a dag G, let $Count(G)$ be the number of total topological orders of G.

Lemma 4 *If a dag G is composed of k static sub-graphs $S_1, S_2, \cdots, S_k$, then*

$$Count(G) = \prod_{i=1}^{k} Count(S_i).$$

Proof: Because S_i, $1 \leq i \leq k$, are static sub-graphs, from Corollary 3, we know that for all topological orders of G, vertices of S_i exclusively occupy a sequence $[|PRED(S_i)|+1, |V|-|SUCC(S_i)|]$. The total number of topological order of S_i is $Count(S_i)$, which are independent from that of $S_j, j \neq i$. According to the fundamental principle of counting, we have $Count(G) = \prod_{i=1}^{k} Count(S_i)$.□

Lemma 5 *If a dag G is composed of k components $G_1, \cdots, G_k$, then*

$$Count(G) = \binom{\sum_{i=1}^{k} |G_i|}{|G_1|, |G_2|, \cdots, |G_k|} \times \prod_{i=1}^{k} Count(G_i).$$

Proof: We can divide the counting of $Count(G)$ into two step: the first step is to count the total number of combination of positions of vertices in $G_i, 1 \leq i \leq k$; the second step is to consider the permutation of vertices within each G_i. For the first step, the problem can be solved by the multinomial coefficient $\binom{\sum_{i=1}^{k} |G_i|}{|G_1|, \cdots, |G_k|}$. For the second step, we can get $\prod_{i=1}^{k} Count(G_i)$, because G_i are independent. Since the two steps are also independent, we have $Count(G) = \binom{\sum_{i=1}^{k} |G_i|}{|G_1|, |G_2|, \cdots, |G_k|} \times \prod_{i=1}^{k} Count(G_i)$. □

Lemma 6 *A disconnected DAG has no static vertex set.*

Proof: We prove this lemma by contradiction. Suppose for an arbitrary disconnected DAG G, which is composed of k components $G_1, G_2, \cdots, G_k$ and it has a static vertex set S. There are two cases:

(1) $S \subseteq V(G_i), 1 \leq i \leq k$

(2) $S = V(G_i) \bigcap V(G_j) \bigcap \cdots \bigcap V(G_k)$

For case (1), because S is a subset of the vertex set of a component G_i, we have $|PRED(S)| + |SUCC(S)| \leq |V(G_i)| - |S| < |V| - |S|$. It is contradictory to the definition of static vertex set.

For case (2), because $V(G_i) \bigcap V(G_j) = \emptyset, i \neq j$, we can have $PRED(S) = \emptyset$ and $SUCC(S) = \emptyset$. Since S is a static vertex set, we have $|S| < |V|$. Therefore, $|PRED(S)| + |SUCC(S)| = 0 < |V| - |S|$. It is also a contradiction to S being a static vertex set.

Therefore, the assumption that S is a static vertex set is not true. □

3 A Divide-and-Conquer Algorithm

The basic idea of the divide-and-conquer algorithm emerges from the results of previous sections. For a DAG having several components, the algorithm will find the count of each component and use the combinatorial formula according to Lemma 5

$$\binom{\sum_{i=1}^{k} |G_i|}{|G_1|, |G_2|, \cdots, |G_k|} \times \prod_{i=1}^{k} Count\ (G_i)$$

to calculate the total where the connected components are $G_1, G_2, \cdots G_k$.

To find the count of a component, the algorithm will identify static vertex sets through which the component might be further partitioned into static sub-graphs and use the combinatorial formula of Lemma 4

$$\prod_{i=1}^{k} Count\ (S_i)$$

to calculate the total with static subgraphs $S_1, S_2, \cdots S_k$. The same algorithm will be recursively applied to each of the sub-DAGs. When a component cannot be further partitioned, counting by enumeration is used.

ORDERS(G) is the default algorithm for counting the number of distinct topological orderings of a graph. Many graphs are counted more efficiently by making use of the properties of static vertex sets, trees, or series parallel graphs.

In each topological ordering the first element will be a vertex with indegree 0. Thus the total number of topological orderings is the sum, over all vertices with indegree 0, of the number of topological orderings having that particular vertex first in the topological ordering. The leads to a recursive formulation of a counting procedure. That is, if F is the set of vertices with indegree 0, then

Algorithm 3.1: Divide-and-Conquer algorithm to count the number of total topological orders of a dag.

```
COUNT (G)
1   if G is connected
2       find static subgraphs S_1, S_2, ... S_k
3       if ( k = 0 ) then return ORDERS(G)
4       else return ∏_{i=1}^{k} COUNT(S_i)
5   else
6       find connected components G_1, G_2, ... G_k
7       return ( Σ_{i=1}^{k} |G_i| over |G_1|, |G_2|, ..., |G_k| ) × ∏_{i=1}^{k} COUNT(G_i)
```

Figure 3: Divide-and-Conquer Algorithm.

$$\text{ORDERS}(G) = \sum_{v \in F} \text{ORDERS}(G \setminus \{v\}),$$

since the set of topological orderings can be partitioned according to the first vertex in the topological order. The following algorithm recursively calculates the number of topological orderings for a graph G.

Algorithm 3.2: Analgorithm to count the number of total topological orders of a dag.

```
ORDERS(G)
/*  Input: Directed Acyclic Graph, G
    Output: The total number of topological orderings of G */
1   if |V(G)| = 1
2       return 1
3   F ← the set of vertices of G with in-degree 0
4   return Σ_{v∈F} ORDERS(G \ {v})
```

Figure 4: A Recursive Algorithm.

The time requirement of the above algorithm is $O(nK)$ where n is the number of vertices in G and K is the number of total topological orders in G.

Theorem 1 *Algorithm 3.1 correctly counts the number of total topological orders of a directed acyclic graph.*

Proof: The correctness of algorithm 3.1 follows directly from lemma 4, lemma 5, and the previous discussion. □

3.1 An Algorithm for Computing Static Vertex Sets

The following definitions will be used in an algorithm for counting the number of topological orderings of graphs containing static sets.

Definition 4 *Given a directed acyclic graph G, the topological interval, $interval[v]$, of a vertex $v \in V(G)$ is represented as $[lower, upper]$, where $lower$ is the lower bound of the position in all the topological orders, and $upper$ is the upper bound of the position in all of topological orders.*

In this paper, a *topological interval* is called an *interval*. We will use $interval[v].lower$ and $interval[v].upper$ to represent the lower bound and the upper bound of the interval of a vertex v.

Definition 5 *Given a DAG G, the span of a vertex $v \in V(G)$ is defined as $span(v) = interval[v].upper - interval[v].lower + 1$.*

Definition 6 *Given a set of vertices S of a DAG G, the span of S is defined as follows: If $min\{interval[v].upper \mid v \in S\} > max\{interval[v].lower \mid v \in S\}$, $span(S) = max\{interval[v].upper \mid v \in S\} - min\{interval[v].lower \mid v \in S\} + 1$. Otherwise, span(S)=0.*

That is to say, the span of a vertex set S can be determined only if the minimum upper bound of all vertices is larger than the maximum lower bound of all vertices. If it is not the case, then the span of S is undefined.

An efficient way is provided to find the static vertex sets of a connected directed acyclic graph. Basically, the idea is to find an interval for a vertex by counting the number of predecessors and successors. The interval of each vertex is determined first. Then the vertices are sorted in an increasing order of the lower bound of the interval. The algorithm scans from the first element of the ordered vertices. Whenever a set of vertices has a span that is equal to the cardinality of the set, a static vertex set is found. The details are shown in Algorithm 3.3. Line 1 builds a transposed dag from the original dag. The time complexity of this line is $O(|V|+|E|)$. Lines 2 to 10 calculate the interval for each vertex by finding the number of predecessors and successors. The time complexity of this step is $O(|V|(|V|+|E|))$. Line 14 sort the interval array in a non-decrement order of the lower bound. If there is a tie between two intervals, the one that has a larger upper bound is put beforehand. The time complexity is $O(|V|log|V|)$. Lines 15 to 40 calculate the static vertex sets. The time complexity is $O(|V|)$. The time complexity is $O(|V|(|V|+|E|))$. The following lemmas prove the correctness of this algorithm.

Lemma 7 *Given a directed acyclic graph G, $\forall v \in V(G)$, $interval[v].lower = |PRED(v)| + 1$, and $interval[v].upper = |V| - |SUCC(v)|$.*

Algorithm 3.3: Find static vertex sets.

```
FindStaticVertexSet( G, M )
1   G' ← ReverseArcs(G);
2   i ← 0;
3   for ( each v ∈ V(G) ) {
4       Reach(v) ← DepthFirstSearch(G, v);
5       Reach'(v) ← DepthFirstSearch(G', v);
6       array[i].lower ← |Reach'(v)| + 1;
7       array[i].upper ← |V(G)| − |Reach(v)|;
8       array[i].vertex ← v;
9       i ← i + 1;
10  }
11  /* array[] is sorted by increasing lower bound. */
12  /* When several element's lower bounds are equal, they are */
13  /* sorted by decreasing upper bound. */
14  array ← Sort(array);
15  i ← 0; count ← 0;
16  l ← array[0].lower; u ← array[0].upper;
17  span ← u − l + 1; last ← l;
18  if (span = |V|) return NULL; /* There is no static vertex set */
19  S ← ∅;
20  P ← ∅;
21  do {
22      P ← P ⋃ array[i].vertex;
23      count ← count + 1;
24      if (array[i].lower > last) {
25          if ( array[i].upper > u ) {
26              u ← array[i].upper;
27              span ← u − l + 1;
28              if (span = |V|) return NULL;
29          }
30          last ← array[i].lower;
31      }
32      i ← i + 1;
33      if ( count = span ) {
34          S ← S ⋃ {P};
35          l ← array[i].lower; u ← array[i].upper;
36          span ← u − l + 1; last ← l;
37          count ← 0;
38          P ← ∅;
39      }
40  } while (i < |V|);
41  return S;
```

Figure 5: Find static vertex sets.

Proof: Given an arbitrary vertex $v \in V(G)$, let $PRED(v)$ be the set of all predecessors of v, and $SUCC(v)$ be the set of all successors of v. For any valid topological order of G, v must be placed after all vertices of $PRED(v)$ and before all vertices of $SUCC(v)$. More over, $\forall w \in V(G) \setminus \{PRED(v) \bigcup SUCC(v) \bigcup \{v\}\}$, w and v are independent, so v can show up anywhere in the interval $[|PRED(v)|+1, |V|-|SUCC(v)|]$. Therefore, the lower bound of $interval(v)$ is $|PRED(v)|+1$, and the upper bound of $interval(v)$ is $|V|-|SUCC(v)|$. □

Lemma 8 *Given a DAG G and a set of vertices $S \subset V(G)$, S is a static vertex set iff $span(S) = |S|$.*

Proof: Sufficient condition: If S is a static vertex set, then according to the definition, for all topological orders of G, all vertices in S show up within a specific interval $[x, y]$. Here x is the smallest lower bound of a vertex in S, y is the largest upper bound of a vertex in S, and $y - x + 1 = |S|$. Therefore, we have $span(s) = |S|$.

Necessary condition: Given an arbitrary set of vertices $S \subset V(G)$, $span(S) = |S|$. Now let $x = min\{interval[v].lower \mid v \in S\}$ and $y = max\{interval[v].upper \mid v \in S\}$, then $y - x + 1 = |S|$. Suppose S is not a static vertex set. Then there are two cases need to be discussed:

(1) S does not appear at the same region within all topological orders.

(2) There exists a proper subset $S' \subset S$, such that, S' is also a static vertex set.

For case (1), there must exist a vertex $v \in S$, such that, v appears before x or after y in a valid topological order, that is, $interval[v].lower < x$ or $interval[v].upper > y$. If $interval[v].lower < x$, $span(S) = y - interval[v].lower + 1 > y - x + 1 = |S|$, which is contradictory to the condition $span(S) = |S|$. If $interval[v].upper > y$, $span(S) = interval[v].upper - x + 1 > y - x + 1 = |S|$, which is also contradictory to the condition $span(S) = |S|$.

For case (2), because S' is a static vertex set, there must exist a vertex $v \in S \setminus S'$, where v appears before all tasks of S', that is, $interval[v].upper < min\{interval[w].lower \mid w \in S'\}$, or v appears after all tasks of S', that is, $interval[v].lower > min\{interval[w].upper \mid w \in S'\}$. Then according to the definition of span of a set, we can get $span(S) = 0$, which is contradictory to $span(S) = |S|$.

We have shown that for both cases, we get an contradiction. Therefore, S is a static vertex set. □

Lemma 9 *Given a DAG G, if there exists a vertex v, such that, $span(v) = |V|$, then G has no static vertex set.*

Proof: If for a DAG G, there is a vertex v, such that, $span(v) = |V|$, then according to the definition, we must have $interval[v].upper = interval[v].lower + |V| - 1$. Because $interval[v].lower \geq 1$, the only possible situation is $interval[v].lower = 1$, $interval[v].upper = |V|$, which means v has no predecessor and no successor. Therefore, v is disconnected from other vertices of G. According to the Lemma 6, G has no static vertex set. □

Theorem 2 *Algorithm 3.3 is correct.*

Proof: The previous lemmas prove the correctness of the necessary steps in algorithm 3.3. Therefore, the algorithm 3.3 is also correct. □

Figure 6 shows an example for finding the static vertex sets of a dag using the algorithm 3.3.

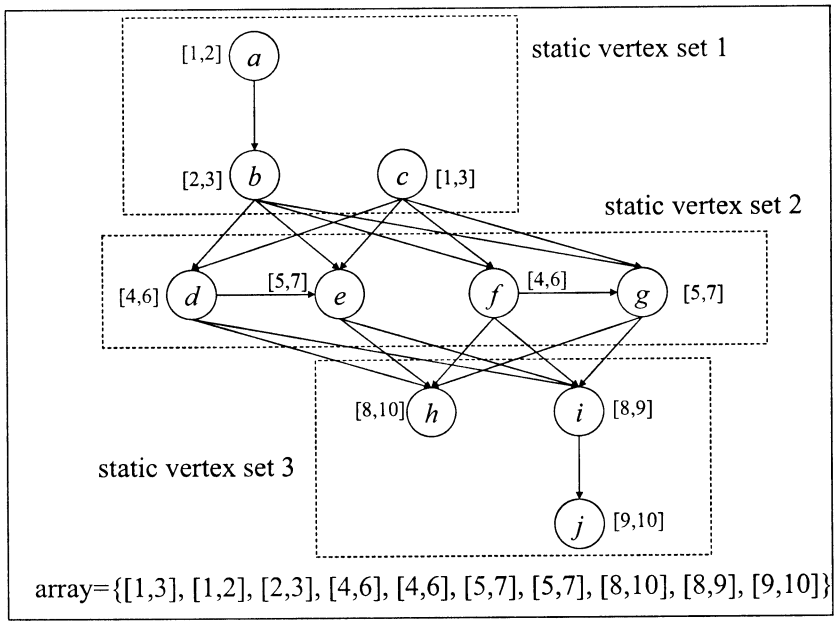

Figure 6: The task graph and its static vertex sets

For a connected subgraph that has no static vertex set, we can use algorithm 3.2 or other exhaustive enumeration methods [4, 2] to count it.

4 Trees and series-parallel DAGs

In general the partitioning process of the divide-and-conquer algorithm may not end up with sub-graphs for which the number of topological orders can be easily determined. For such sub-graphs, enumeration algorithms are used to determine the number of topological orders. For DAGs satisfying certain structural properties, such as trees and series-parallel DAGs, a complete partitioning is possible and no enumeration is needed. We analyze the situations when the input DAG of the algorithm is a tree or a series-parallel graph in this section. In fact, a complete partitioning can be achieved whenever the components resulting from the static sub-graph partitioning are trees and series-parallel graphs.

Whether a given DAG is a tree can be easily recognized in linear time. The recognition algorithm will produce a tree structure whenever the input DAG is a tree. In such a tree structure, the root vertex is a static vertex, which partitions the rest of the tree into components. Each component is a tree by itself. Hence, combinatorial equations of lemmas 4 and 5 can be applied recursively. Using results of previous sections, a post order traversal of the tree (input DAG G), which can be done in linear time, will give us $Count(G)$. Notice that the relative expensive step of computing the static vertex set is eliminated in this case.

Series-parallel digraph is defined recursively [3]. For the basis, a single vertex graph is a series-parallel digraph. For the recursive steps, a series-parallel is constructed from two series-parallel digraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ through either parallel composition or series composition. In parallel composition, the new graph is $G_p = (V_1 \bigcup V_2, E_1 \bigcup E_2)$. In series composition, the new graph is $G_s = (V_1 \bigcup V_2, E_1 \bigcup E_2 \bigcup (T_1 \times S_2))$, where T_1 is the set of sinks of G_1 and S_2 is the set of sources of G_2.

It is known that the recognition of whether a input DAG is a series-parallel graph can be done in linear time [3]. The recognition algorithm will produce a binary decomposition tree representing the series-parallel digraph. In the decomposition tree, each leaf node is a vertex and each non-leaf node has a label 'S' for the series composition or a 'P' for the parallel composition. Notice that vertices involved in a series composition form a static vertex set, and subgraphs involved in a parallel composition are components. Hence, results of lemma 4 and lemma 5 can again be used, as in the tree structure. In almost an identical fashion, the post order traversal of the decomposition tree, which can be done in linear time, will give us $Count(G)$ when G is a series-parallel digraph.

Notice that for an input DAG that is a tree or a series-parallel digraph, algorithm 3.1 always produces a complete partitioning. However, the recognition of trees and series-parallel graphs is in linear time, and thus the aforementioned traversal technique may further improve algorithm 3.1.

The new algorithm is shown in Figure 7.

Algorithm 3.4: An improved divide-and-Conquer algorithm to count the number of total topological orders of a dag.

```
CountNew (G)
1   if G is a tree
2       return CountTree(G)
3   if G is a series-parallel digraph
4       return CountSeriesParallelDigraph(G)
5   if G is connected
6       find static subgraphs S_1, S_2, ... S_k
7       if ( k = 0 ) then return Orders(G)
8       else return ∏_{i=1}^{k} Count(S_i)
9   else
10      find connected components G_1, G_2, ... G_k
11      return ( Σ_{i=1}^{k} |G_i| / |G_1|, |G_2|, ..., |G_k| ) × ∏_{i=1}^{k} Count(G_i)
```

Figure 7: An improved divide-and-Conquer Algorithm.

5 Experimental results

To study of the effectiveness of the proposed algorithm, we have implemented the algorithm in C++ and conducted a few preliminary test runs on some randomly generated dags. A total enumeration algorithm reported in [2] has also been implemented in C++ for comparison.

The table in Figure 8 shows the run time of the two algorithms. As can be seen from the table, the divide and conquer scheme is much faster than the enumeration scheme. The speed up fluctuates greatly from 10 to 10^{14}, depending on the edge density and the structure of the input DAG. For some test cases, in which the DAGs have more than 20 vertices, run time is estimated because the number of topological orders is so large that the enumeration algorithm cannot finish in a reasonable amount of time. Since a simple linear relationship exists between the number of topological orders and the time to do enumeration, the estimations, shown by * in Figure 8, are believe to be pretty accurate.

Let us consider the test case 8 as an example. The corresponding DAG is shown in Figure 9. The DAG consists of 17 vertices and 19 edges. The total topological orders are 891,928,800. It takes the enumeration program 57,235 milliseconds to count the number, where as the divide conquer program only spends less than 1 millisecond to do that. The difference between

the performance of the two schemes is due to the structure of the DAG. The DAG contains several components, the divide and conquer algorithm takes advantage of that and uses the combinatorial equation to the count. As can be seen from figure 9, the DAG G has four components, $C_1 = \{0\}, C_2 = \{12\}, C_3 = \{5, 16\}$, and $C_4 = \{1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13, 14, 15\}$. Notice that the component C_4 has no static vertex set and the enumeration algorithm is used to count the total topological orders of C_4, which turn out to be 31,230. The total topological orders, $Count(G)$, of G can be calculated by the following equation:

$$Count(G) = \frac{|V(G)|!}{\prod_{i=1}^{k} |V(G_i)|!} \times \prod_{i=1}^{k} Count(G_i) = \frac{17!}{13! \times 2! \times 1! \times 1!} \times 31230$$

The final result of the $Count(G)$ is 891,928,800, as shown in Figure 8. The results presented here are preliminary. A more thorough experimentation is warranted to study some of issues further.

6 Conclusions and Open Issues

In this paper, we have proposed a divide-and-conquer method that partitions certain types of DAG into sub-graphs from which the number of topological orderings is calculated using combinatorial methods, given the number of topological orderings of the subgraphs. The concepts of static vertex and static vertex set were introduced and used in the partitioning of the DAG into sub-graphs. An algorithm has been proposed to identify static vertices and static vertex sets, and their induced sub-graphs, called static sub-graphs. Preliminary experiments have been conducted to compare the effectiveness of the proposed algorithm for counting the number of topological orders to a standard algorithm for enumerating topological orders.

In the future we hope to expand the characterization of DAGs for which techniques, similar to those developed in this paper, will yield efficient methods for the calculation of the number of distinct topological orderings. Other methods such as dynamic programming will be considered to see if the time complexity of the algorithm to compute ORDERS(G) might be improved. Last but not the least, more experimentation will be conducted to study the performance of the proposed algorithm on various types of dags.

Time Complexity of the Counting Algorithms

Test Case	Vertex Number	Edge Number	Total Topological Orders	Divide and Conquer (ms)	Enumeration (ms)	
1	10	6	39600	0	0	
2	11	7	166320	0	16	
3	12	7	2882880	0	172	
4	13	10	6486480	0	359	
5	14	12	23783760	0	1469	
6	15	18	19688130	16	1156	
7	16	17	686415360	1031	38938	
8	17	19	891928800	0	57235	
9	18	24	506216502	2672	31203	
10	19	32	167712696	94	9985	
11	20	29	11135900000	5235	706312	
12	21	51	1604451096	7531	91937	
13	22	19	1.71533E+16	4953	1.00717E+12	*
14	23	62	35339431552	167656	2091067	
15	24	10	6.59183E+18	0	3.92453E+14	*
16	25	103	9183600	16	531	

Figure 8: Time complexity of the counting algorithms

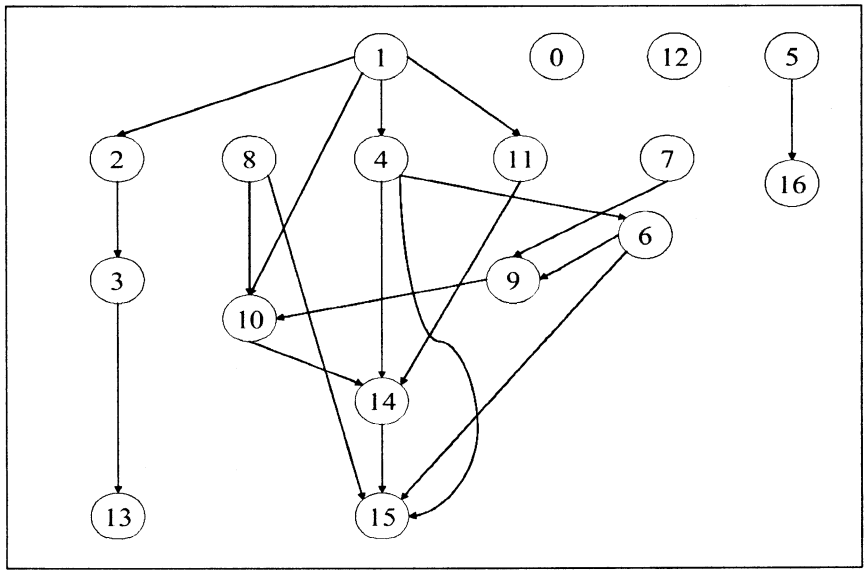

Figure 9: A randomly generated DAG.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms.* McGraw-Hill, 2 edition, 2001.

[2] A. D. Kalvin and Y. L. Varol. On the generation of all topological sortings. *Journal of Algorithms*, 4:150–162, 1983.

[3] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11:298–313, 1982.

[4] Y. L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24:83–84, 1981.

[5] Zhichun Xiao. *On Cluster Scheduling Algorithms.* PhD thesis, University of Arkansas, Fayetteville, Computer Science and Computer Engineering Department, 2005.