TCS-TR-A-14-75

# TCS Technical Report

# An Efficient Method of Indexing All Topological Orders for a Given DAG

**by**

Yuma Inoue and Shin-ichi Minato

**Division of Computer Science**

**Report Series A**

July 17, 2014

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682
Fax: +81-011-706-7682

# An Efficient Method of Indexing All Topological Orders for a Given DAG

Yuma Inoue
Division of Computer Science
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

Shin-ichi Minato*
Division of Computer Science
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

July 17, 2014

**Abstract**

Topological orders of a directed graph are an important concept of graph algorithms. The generation of topological orders is useful for designing graph algorithms and solving scheduling problems. In this paper, we generate and index all topological orders of a given graph. Since topological orders are permutations of vertices, we can use the data structure $\pi$DD, which generates and indexes a set of permutations. In this paper, we propose *Rot-$\pi$DD*s, which are a variation of $\pi$DDs based on a different interpretation. Compression ratios of Rot-$\pi$DDs for representing topological orders are theoretically improved from the original $\pi$DDs. We propose an efficient method for constructing a Rot-$\pi$DD based on dynamic programming approach. Computational experiments show the amazing efficiencies of a Rot-$\pi$DD: a Rot-$\pi$DD for $3.7 \times 10^{41}$ topological orders has only $2.2 \times 10^7$ nodes and is constructed in 36 seconds. In addition, the indexed structure of a Rot-$\pi$DD allows us to fast post-process such as edge addition and random samplings.

## 1 Introduction

Topological sort is one of the classical and important concepts of graph algorithms. Vertex orders obtained by topological sort are used to analyze characteristics of a directed graph structure and support graph based algorithms [6]. Furthermore, topological orders are equivalent to linear extensions of a poset, i.e., total orders which are in no contradiction with the partially ordered set defined by directed edges of a graph. Thus, topological sort plays an important role in several research areas such as discrete mathematics and computer science, and has many applications such as graph problems and scheduling problems [14].

*He also works for JST ERATO Minato Project.

Linear time algorithms calculating a topological order are classical and well-known algorithms, and dealt with by Cormen et al. [6]. In recent researches, two derived problems are mainly discussed. One of these is an online topological sort, i.e., calculation of a topological order on a dynamic graph. Bender et al. [2] proposed a topological sort algorithm which allows edge insertions, and Pearce et al. [13] proposed an algorithm which can also handle edge deletions. Another one is the enumeration problem of all topological orders. Ono et al. [12] presented a worst case constant delay time generating algorithm using family trees. The complexity of the counting problem has been studied from several aspects since Brightwell et al. [3] proved that it is $\#P$-complete. Bubley et al. [4] proposed a randomized algorithm to approximate the number of all linear extensions. Li et al. [10] provided an experimentally fast algorithm counting all topological orders based on Divide & Conquer method. There are many polynomial time counting algorithms when we restrict the graph structure or fix some graph parameters, e.g., trees and bounded poset width [1, 5].

In this paper, we deal with both of these problems. That is, our goal is generation of all topological orders of given graphs and manipulation of these orders when the graph is dynamically changed, e.g., edge addition. In addition, we implicitly store all topological orders as a compressed data structure in order to handle graphs that are as large as possible. Experimental results, which will be described later, show that our algorithm and data structure work very well: $3.7 \times 10^{41}$ topological orders of a directed graph with 50 vertices are generated in 36 seconds, and the compressed data size is only about 1 gigabyte. Furthermore, an edge addition query for a directed graph with 25 vertices is done in 1 second.

Our method is based on an indexed data structure compactly representing a set of permutations, *permutation decision diagram*, also called *$\pi$DD* or *PiDD* [11]. Although a $\pi$DD can be used to achieve our purpose, compression ratio and query processing are not efficient enough practically or theoretically. Thus, we developed a new variation of $\pi$DD, named *Rot-$\pi$DD* (*Rotation-based $\pi$DD*). The key idea of our modification is a direct construction of a decision diagram based on the dynamic programming approach. This modification realizes the practical efficiency of compression and query processing, which are also bounded theoretically.

Our contributions in this paper are summarized as follows.

- We provide the first algorithm for implicit generation of all topological orders with dynamic manipulation.
- Time and space for construction and query processing of our algorithm are efficient experimentally and theoretically, while it is difficult to estimate the size and computation time of decision diagrams in general.

The rest of this paper is organized as follows. Section 2 introduces a precise definition of topological sort and algorithms for counting, which will be used in our algorithm. Section 3 introduces $\pi$DDs and our modified version (Rot-$\pi$DDs) for

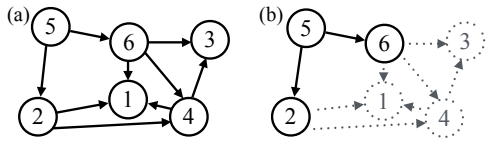


Figure 1: (a) A DAG and (b) the subgraph induced by the vertex set $\{2, 5, 6\}$.

generation of all topological orders. Our algorithm for construction of a Rot-$\pi$DD is also presented in Section 3. In Section 4, we prove the theoretical bound of the time complexity of our algorithm and the size of the new permutation decision diagram for all topological orders. Section 5 presents experimental results of generation and query processing, comparing with existing $\pi$DD and other existing methods. Section 6 gives some consequences of this paper.

## 2 Topological Orders

We define a *directed graph* $G = (V, E)$, where $V$ is a vertex set and $E$ is a directed edge, i.e., $E \subseteq \{(u, v) \mid u, v \in V\}$. Note that $(u, v)$ is an ordered pair of two vertices. Let $n$ be the number of vertices and $m$ be the number of edges. Without loss of generality, we can assume $V = \{1, 2, \ldots, n\}$.

A *topological order* of a graph $G$ is an ordering $v_1 v_2 \ldots v_n$ of all vertices such that $v_i$ must precede $v_j$ if $(v_i, v_j) \in E$. For example, the graph in Fig. 1(a) has four topological orders: 526413, 526431, 562413, and 562431.

A directed graph is a *DAG* (*Directed Acyclic Graph*) if the graph has no cycle. In this paper, we assume that given graphs are DAGs because we can determine whether or not a graph has cycles in linear time, and if so, there is no topological order.

There are many linear time algorithms for computing a topological order of a given graph [9, 15]. One of the key ideas is deleting vertices whose out-degree is 0. If there is no edge from $v$, $v$ can be the rightmost element in a topological order, because there is no element that must be preceded by $v$. We delete such $v$ and its incident edges, i.e., after the deletion of $v$, we can consider only the subgraph induced by the vertex subset $V \setminus \{v\}$. Then, we repeat the same procedure for the induced subgraph and obtain a topological order of the induced subgraph recursively. Finally, we concatenate a topological order of the induced subgraph and $v$ to obtain a topological order of the given graph. The time complexity of this algorithm is $O(n + m)$.

Similarly, an algorithm counting all topological orders of a given graph can be designed recursively. Let $G(X)$ denote the subgraph of $G$ induced by the vertex subset $X$. For each recursion, we assume that the current vertex subset is $V'$. Then, for each vertex $v$ whose out-degree is 0 in $G(V')$, we sum up the numbers of all topological orders of $G(V' \setminus \{v\})$. The time complexity of this algorithm is

$O((n+m)TO(G))$, where $TO(G)$ is the number of the topological orders of the graph $G$. Since $TO(G) = O(n!)$, the time complexity is $O((n+m)n!)$. We can improve this complexity by a *dynamic programming* (*DP*) approach.

For example, in Fig. 1(a), we can delete vertices $\{1,3,4\}$ in the order 134 or 314. (Note that a deletion order is the reverse of a topological order.) Then we obtain the same induced subgraph on $\{2,5,6\}$. Although $TO(G(\{2,5,6\}))$ is not changed, we redundantly count $TO(G(\{2,5,6\}))$ in each recursion of 134 and 314. Thus, by memorizing the calculation result $TO(G(V'))$ for $G(V')$ at the first calculation, we can avoid duplicated calculations for each $G(V')$. In other words, this is a top-down DP, which recursively calculates $TO(G(V')) = \sum_{v \in V'_0} TO(G(V' \setminus \{v\}))$, where $V'_0$ is the set of vertices whose out-degree is 0 in $G(V')$. We define *valid induced subgraphs* of $G$ as induced subgraphs $G(V')$ that can appear in the above DP recursion. Let $IS(G)$ denote the number of valid induced subgraphs of $G$. Then, this DP algorithm uses $O((n+m)IS(G))$ time and $O(IS(G))$ space. In the worst case, $IS(G) = 2^n$, which is the number of all subsets of $V$. Therefore, we improve the complexity from factorial $O((n+m)n!)$ to exponential $O((n+m)2^n)$.

The idea of valid induced subgraphs is equivalent to upsets in a poset in the talk of Cooper [5]. Cooper provided another upper bound $O(n^w)$ of $IS(G)$, where $w$ is the width of a poset corresponding to $G$. The proof of this bound and more precise analyses will be described in Section 4.

Here, we remember our goal in this paper again. Our goal is generating and indexing all topological orders, which are permutations of vertices. Thus, it is reasonable to expect that a compressed and indexed data structure for permutations can be useful for this purpose. And if we can compress permutations in the same way as the above DP, the compression size is bounded by $IS(G) = O(\min\{2^n, n^w\})$, which can be quite smaller than $TO(G)$.

## 3 Permutation Decision Diagrams

In this section, we introduce a compressed and indexed data structure for permutations, $\pi$DD, and discuss whether or not compression of a $\pi$DD is suitable for the DP approach.

### 3.1 Existing Permutation Decision Diagrams: $\pi$DDs

First, we define some notations about permutations. A *permutation* of length $n$, or $n$-permutation, is a numerical sequence $\pi = \pi_1\pi_2\dots\pi_n$ such that all elements are distinct and $\pi_i \in \{1,2,\dots,n\}$ for each $i$. The identity permutation of length $n$ is denoted by $e^n$, which satisfies $e^n_i = i$ for each $1 \leq i \leq n$.

We define a *swap* $\tau_{i,j}$ as the exchange of the $i$th element and the $j$th element. Any $n$-permutation can be uniquely decomposed into a sequence of at most $n-1$ swaps. This swap sequence is defined as the series of swaps to obtain an objective

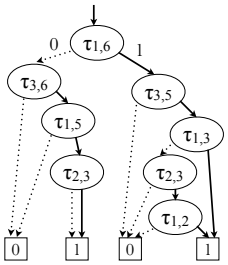


Figure 2: The $\pi$DD representing $\{526413, 526431, 562413, 562431\}$.

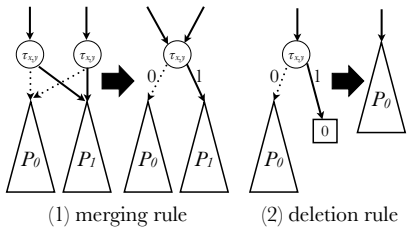


Figure 3: Two reduction rules of $\pi$DDs.

$n$-permutation $\pi$ from the identity permutation $e^n$ by a certain algorithm. The algorithm repeats swaps to move $\pi_k$ to the $k$th position, where $k$ runs from right to left. For example, we consider a decomposition of the permutation $\pi = 43152$ into a swap sequence. We start with $e^5 = 12345$. The 5th element of $\pi$ is 2 and 2 is the 2nd element of $e^n$, hence we swap the 2nd element and the 5th element, and obtain $15342 = \tau_{2,5}$. Next, since the 4th element of $\pi$ is 5, and 5 is the 2nd element, we then obtain $14352 = \tau_{2,5} \cdot \tau_{2,4}$. Repeating this procedure, we finally obtain $\pi = 43152 = \tau_{2,5} \cdot \tau_{2,4} \cdot \tau_{1,3} \cdot \tau_{1,2}$.

A *$\pi$DD* is a data structure representing a set of permutations canonically [11], and has efficient set operations for permutation sets. $\pi$DDs consist of five components: nodes with a swap label, 0-edges, 1-edges, the 0-sink, and the 1-sink. Fig. 2 shows the $\pi$DD representing topological orders of the graph in Fig. 1(a).

Each internal node has exactly a 0-edge and a 1-edge. Each path in a $\pi$DD represents a permutation: if a 1-edge originates from a node with label $\tau_{x,y}$, the decomposition of the permutation contains $\tau_{x,y}$, while a 0-edge from $\tau_{x,y}$ means that the decomposition excludes $\tau_{x,y}$. If a path reaches the 1-sink, the permutation corresponding to the path is in the set represented by the $\pi$DD. On the other hand, if a path reaches the 0-sink, the permutation is not in the set.

A $\pi$DD becomes a compact and canonical form by applying the following two reduction rules (Fig. 3):

(1) Merging rule: share all nodes which have the same labels and child nodes.

(2) Deletion rule: delete all nodes whose 1-edge points to the 0-sink.

Although the size of a $\pi$DD (i.e. the number of nodes in a $\pi$DD) can grow exponentially ($O(2^{n^2})$) with respect to the length of permutations, in many practical cases, $\pi$DDs demonstrate high compression ratio. In addition, $\pi$DDs support efficient set operations such as union, intersection, and set difference. The computation time of $\pi$DD operations depends on the size of $\pi$DDs, not on the number of permutations in the sets represented by the $\pi$DDs.

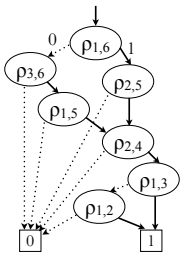


Figure 4: The Rot-$\pi$DD representing $\{526413, 526431, 562413, 562431\}$.

### 3.2 DP Approach and $\pi$DDs

Now, we consider whether or not we can directly construct a $\pi$DD in the same way as the DP approach described in Section 2.

Here, we note that the swap decomposition algorithm behaves as deletions of a vertex on an induced subgraph. We can represent the current recursive state in DP procedure as a permutation, i.e., let $k$ be the number of vertices of the current induced subgraph, then the $k$-prefix of an $n$-permutation represents the vertex set of the induced subgraph, and the $(n-k)$-suffix of the permutation represents the reverse order of deletions. Furthermore, a deletion of a vertex $v$ can be described as a swap $\tau_{i,k}$, where $i$ is the position of $v$ in the permutation. For example, we can consider a permutation 625431 represents the subgraph in Fig. 1(b) such that the deletion order is 134. When we delete the vertex 6, we swap the 1st position, which is 6, and the 3rd position, which is the rightmost of the $k$-prefix representing the vertex subset. Then, we obtain 526431, which represents the subgraph induced by $\{2, 5\}$ and the reverse order of deletions.

By compressing swap sequences into a $\pi$DD, we can recursively construct a $\pi$DD for all topological orders. That is, for each recursion represented as a permutation $\pi$, if we apply $\tau_{i,j}$ to delete $\pi_i$, we create the new $\pi$DD such that its root node is $\tau_{i,j}$, its 1-edge child is the $\pi$DD for swap sequences after applying $\tau_{i,j}$, and its 0-edge child is the $\pi$DD for swap sequences in which we do not apply $\tau_{i,j}$. The $\pi$DDs for the 1-edge and 0-edge child are recursively constructed.

However, deletions by swaps are not available for DP. In order to use DP approach, swap sequences for the same induced subgraph must be uniquely determined. Even if different prefixes of permutations represent the same induced subgraph, their swap sequences can differ. For example, consider the DAG in Fig. 1. Deletion sequences 314 and 134 generate the same induced subgraph on $\{2, 5, 6\}$, and these states are represented as 526413 and 625431, respectively. The induced subgraph on $\{2, 5, 6\}$ has a topological order 526. In order to obtain this, we apply no swap to 526413, while we apply $\tau_{1,3}$ to 625431. This means there are multiple $\pi$DDs corresponding to the same induced subgraph.

**Algorithm 1** Rot-$\pi$DD construction for all topological orders of $G = (V, E)$.

*ConstructRotPiDD*($G$):
**if** V is empty **then**
  **return** 1-sink
**else if** have never memorized the Rot-$\pi$DD $R_G$ for $G$ **then**
  Rot-PiDD $R \leftarrow$ 0-sink
  **for** each $v$ whose out-degree is 0 in $G$ **do**
    Integer $i \leftarrow v$'s position in the increasing sequence of $V$, $j \leftarrow |V|$
    $R \leftarrow$ the Rot-PiDD with root node $\rho_{i,j}$, left child $R$, and right child *ConstructRotPiDD*($G(V \setminus \{v\})$)
  **end for**
  memorize $R$ as $R_G$
**end if**
**return** $R_G$

### 3.3 New Permutation Decision Diagrams: Rot-$\pi$DDs

As described in the previous subsection, the DP approach cannot be used to directly construct a $\pi$DD. To overcome this problem, we use another decomposition where each vertex subset is uniquely represented as a prefix of permutations. In order to realize this, we use the left-rotation decomposition. A *left-rotation* $\rho_{i,j}$ rearranges $i$th element into $j$th position, and $k$th element into $(k-1)$th position for each $i + 1 \leq k \leq j$. That is, $\rho_{i,j}$ rearranges an $n$-permutation $\pi_1 \dots \pi_i \pi_{i+1} \dots \pi_j \dots \pi_n$ into $\pi_1 \dots \pi_{i+1} \dots \pi_j \pi_i \dots \pi_n$.

Left-rotations also can uniquely decompose a permutation. The left-rotation decomposition is similar to the one for swaps: we start with $e^n$ and repeatedly apply $\rho_{i,j}$ to move $\pi_i$ to the $j$th position, from right to left. For example, consider to decompose 43152 into a sequence of left-rotations. We start with $e^5 = 12345$. Now, we move 2 from the 2nd position to the 5th position. Thus, we obtain $13452 = \rho_{2,5}$. Next, we move 5 from the 4th position to the 4th position, i.e., we do not rotate. Repeating this procedure, we finally obtain $43152 = \rho_{2,5} \cdot \rho_{1,3} \cdot \rho_{1,2}$.

Left-rotations realize the unique representation of an induced subgraph as a prefix of a permutation, because a prefix is always in an increasing order. Left-rotation $\rho_{i,j}$ only changes the relative order between the $i$th element and the elements in $[i+1, j]$, i.e., relative orders in $[1, j-1]$ are not changed. This means the $(j-1)$-prefix is always in increasing order when we start with $e^n$ and apply $\rho_{i,j}$ in decreasing order of $j$.

Thus, we can use the DP approach by using left-rotations as node labels of $\pi$DDs. We call this left-rotation based $\pi$DD *Rot-$\pi$DD*, and existing $\pi$DD *Swap-$\pi$DD* to distinguish. Fig. 4 illustrates the Rot-$\pi$DD for the same set as Fig. 2. Algorithm 1 describes the DP based construction algorithm of a Rot-$\pi$DD.

### 3.4 Rot-$\pi$DD operations

Since Rot-$\pi$DDs are decision diagrams, they can use the same set operations as Swap-$\pi$DD such as union, intersection, and set difference. Some queries such as random samplings and counting the cardinality of the set represented by a Rot-$\pi$DD are also available without any modification. On the other hand, some queries have to be redesigned. For example, the precedence query $R.Precede(u, v)$ returns the Rot-$\pi$DD that represents only permutations $\pi$ extracted from the Rot-$\pi$DD $R$ such that $u$ precedes $v$ in $\pi$. This query is equivalent to addition of the edge $(u, v)$ in a graph. This query can be designed as a recursive procedure (we omit details), and the runtime depends on only the size of the Rot-$\pi$DDs.

## 4 Theoretical Analysis

In this section, we analyze the time and the space complexity of DP based counting and Rot-$\pi$DD construction. Here, we remember the definition of $IS(G)$: $IS(G)$ is the number of the induced subgraphs of $G$ that can be obtained by deletions of vertices with out-degree 0. We start by proving the bound $O(n^w)$ of $IS(G)$. According to Dilworth's theorem [7], the width $w$ of a poset equals the *minimum path cover* of the DAG corresponding to the poset, where a path cover of a graph $G$ is a set of paths in $G$ such that each vertex of $G$ must appear in at least one of the paths. Therefore, it is sufficient to prove the following theorem.

**Theorem 1.** *Given a DAG $G$ with $n$ vertices and minimum path cover $w$, $IS(G) \leq (n+1)^w$ holds.*

*Proof.* Let $p_i$ be the $i$th path of the minimum path cover and $l_i$ be the length of $p_i$. Here, all vertices in a valid induced subgraph must be consecutive in prefix of each $p_i$ due to precedence. The number of the possible prefixes of each path is at most $l_i+1$, and the number of paths is $w$. Therefore, $IS(G)$ is bounded by $\prod_{k=1}^{w}(l_k+1)$. Since $l_i$ is also bounded by $n$, $IS(G) \leq (n+1)^w$ holds. □

In this proof, we use the rough estimation $l_i = n$, but in fact $\sum_{k=1}^{w} l_k = n$ holds. We can prove a tighter bound using this restriction.

**Lemma 2.** *If $\sum_{k=1}^{w} l_k = n$ holds, $\prod_{k=1}^{w}(l_k+1) \leq (n/w+1)^w$ holds for all positive integers $n$, $1 \leq w \leq n$, and $1 \leq l_i \leq n$.*

*Proof.* The proof is by induction. We prove it in the Appendix. □

**Corollary 3.** *Given a DAG $G$ with $n$ vertices and minimum path cover $w$, $IS(G) \leq (n/w+1)^w$ holds.*

*Proof.* The proof follows from the proof of Theorem 1 and Lemma 2. □

Corollary 3 gives a new bound of $IS(G)$. Since $(n/w+1)^w$ is monotonically nondecreasing for all positive integers $n$ and $w$, the range of $(n/w+1)^w$ is $[n+1, 2^n]$ for $1 \leq w \leq n$. This means the previous bound $O(\min\{2^n, n^w\})$ can be directly replaced by $O((n/w+1)^w)$. Hence, we obtain the time complexity $O((n+m)(n/w+1)^w)$ and the space complexity $O((n/w+1)^w)$ of the DP.

We can also estimate the size of a Rot-$\pi$DD representing all topological orders and the time of the construction. The size of such a Rot-$\pi$DD is at most $w$ times larger than the space of DP because each DP recursion has at most $w$ transitions, while each node of a Rot-$\pi$DD has exactly two edges. Therefore, the size of such a Rot-$\pi$DD is at most $O(w(n/w+1)^w)$. [1] On the other hand, the time of the construction is as fast as DP, because each node is only created for each vertex deletion in constant time. Hence, the time complexity of the construction of a Rot-$\pi$DD representing all topological orders is $O((n+m)(n/w+1)^w)$.

## 5 Computational Experiments

We measured the performance of our Rot-$\pi$DD construction algorithm by computational experiments. Experiment setting is as follows.

- Input: A DAG.
- Output: The number of topological orders of the given DAG.
- Test Cases: For each $n = 5, 10, 15, \ldots, 45, 50$ and $k = 1, 3, 5, 7, 9$, we generate exactly 30 random DAGs with $n$ vertices and $\lfloor \frac{k}{10} \times \frac{n(n-1)}{2} \rfloor$ edges. (That is, $k$ provides the edge density of DAGs.)

We also compared with other methods on the same setting. Comparisons are Swap-$\pi$DD construction, DP counting, and Divide & Conquer counting [10]. Since direct construction of a Swap-$\pi$DD is inefficient, we apply precedence queries for each edge individually. We implemented all algorithms in C++ and carried out experiments on a 3.20 GHz CPU machine with 64 GB memory.

Fig. 5 and Fig. 6 show the average runtime and memory usage on $n = 20$ cases. Divide & Conquer method times-out on some cases of $k = 1$. These results indicate that the worse cases of all algorithms are sparse graphs. In general, sparse graphs tend to have a large poset width. In fact, the average $w$ of $k = 1$ cases is 10.6, while that of $k = 5$ cases is 3.3. Therefore, the complexity $O((n/w+1)^w)$ also tends to become large on the sparse graph cases.

We therefore focus on sparse graphs. Table 1 shows the average numbers of topological orders, the sizes of Rot-$\pi$DDs, and runtimes on the case $k = 1$. It shows the amazing efficiency of Rot-$\pi$DDs: $3.7 \times 10^{41}$ topological orders are compressed

[1] Note that this bound is valid only for all topological orders. For any permutation set, the worst size of Rot-$\pi$DDs is $O(2^{n^2})$, which is same as the size bound of Swap-$\pi$DDs.

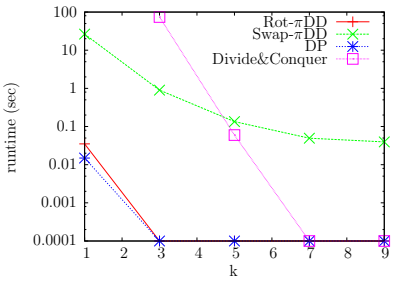


Figure 5: Average runtime for construction when $n = 20$.

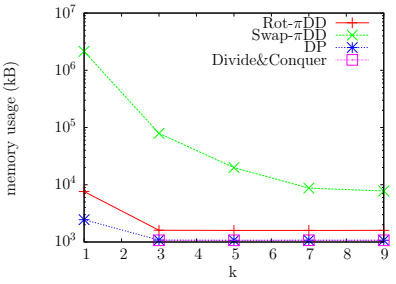


Figure 6: Average memory usage for construction when $n = 20$.

Table 1: Experimental results on the cases $k = 1$.

| $n$ | The number of topological orders | Rot-$\pi$DD size | Time (sec) |
|---|---|---|---|
| 5 | 60 | 16 | 0.00 |
| 10 | 270816 | 310 | 0.00 |
| 15 | 3849848730 | 3990 | 0.00 |
| 20 | 84248623806362 | 35551 | 0.04 |
| 25 | 1729821793136903967 | 179205 | 0.18 |
| 30 | 166022551499377802024339 | 695029 | 0.90 |
| 35 | 188972608055858740408591896 98 | 2634015 | 3.78 |
| 40 | 19224622437706527112568934998 0187 | 4649639 | 6.68 |
| 45 | 75068589270080843845910704526224 56252 | 8288752 | 12.69 |
| 50 | 37563660779499151811427427955995243 1497225 | 22542071 | 35.51 |

into a Rot-$\pi$DD that has only $2.2 \times 10^7$ nodes in 36 seconds on the case $n = 50$. Note that each node of Rot-$\pi$DDs consumes about 30 bytes.

Fig. 7 and Fig. 8 show the average runtime and memory usage on $k = 1$ cases. Swap-$\pi$DD and Divide & Conquer time-out on the case $n \geq 25$ and $n \geq 20$, respectively. We can obtain a Rot-$\pi$DD, which supports many operations for queries, with only tenfold increase in runtime and memory usage compared to DP. We guess that the overhead time is used to store new nodes of a Rot-$\pi$DD into the hash table, and the overhead memory is caused by the difference of the space complexities between DP and Rot-$\pi$DD as described in Section 4.

We also carried out experiments to measure the performance of query processing. On these experiments, we use 30 random DAGs with 25 vertices and 90 edges. We start with a graph having no edge, and add each edge individually. The Rot-$\pi$DD method uses precedence queries for each edge addition, while DP recomputes $TC(G)$ for each addition. We measure the runtime and the size of a Rot-$\pi$DD and a DP table. Note that the DP table size equals $IS(G)$.

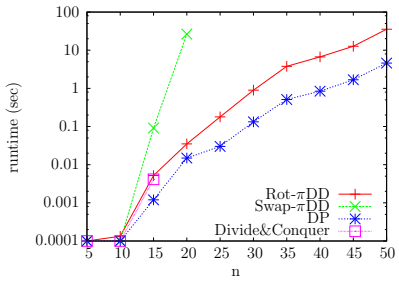


Figure 7: Average runtime for construction when $k = 1$.

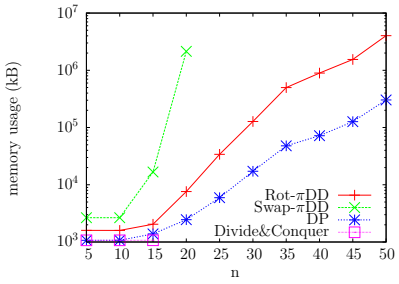


Figure 8: Average memory usage for construction when $k = 1$.

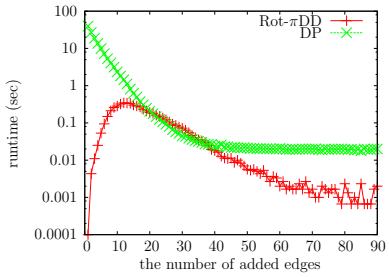


Figure 9: Average runtime for edge addition queries.

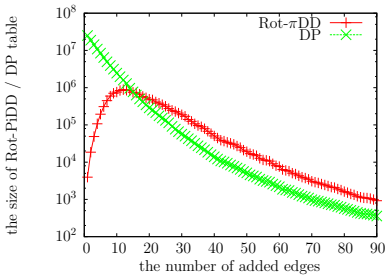


Figure 10: Average space for edge addition queries.

Fig. 9 and Fig. 10 show the results for query processing. In almost all cases, Rot-$\pi$DDs can generate and index all topological orders faster than or equal to DP. Especially in sparse cases, query processing of Rot-$\pi$DDs is very efficient. It may be because Rot-$\pi$DDs (and Swap-$\pi$DDs) can represent the set of all $n$-permutations with $n(n-1)/2+1$ nodes (please refer to [8] for more details).

## 6 Conclusion

In this paper, we gave an efficient method for generating and indexing all topological orders of a given DAG. We proposed a new data structure Rot-$\pi$DD, which is suitable for indexing topological orders. Theoretical analysis and experiments showed the efficiency of our construction algorithm, compression ratios of Rot-$\pi$DDs, and query processing.

Future work is to apply Rot-$\pi$DDs to solve several scheduling problems. We would like to develop new operations to process required queries and optimizations for each problem. Another topic is to apply the Rot-$\pi$DD construction technique to other graph generation problems such as Hamiltonian paths and perfect elimination orderings. These problems can also be recursively divided into subproblems based on induced subgraphs.

## References

[1] Mike D Atkinson. On computing the number of linear extensions of a tree. *Order*, 7(1):23–25, 1990.

[2] Michael A Bender, Jeremy T Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In *20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1108–1115. Society for Industrial and Applied Mathematics, 2009.

[3] Graham Brightwell and Peter Winkler. Counting linear extensions. *Order*, 8(3):225–242, 1991.

[4] Russ Bubley and Martin Dyer. Faster random generation of linear extensions. *Discrete Mathematics*, 201(1):81–88, 1999.

[5] Joshua N Cooper. When is linear extensions counting easy? *AMS Southeastern Sectional Meeting*, 2013.

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, 2001.

[7] Robert P Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

[8] Yuma Inoue. Master's thesis: Generating PiDDs for indexing permutation classes with given permutation patterns. Technical Report TCS-TR-B-14-9, Division of Computer Science, Hokkaido University, 2014.

[9] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[10] Wing-Ning Li, Zhichun Xiao, and Gordon Beavers. On computing the number of topological orderings of a directed acyclic graph. *Congressus Numerantium*, 174:143–159, 2005.

[11] Shin-ichi Minato. $\pi$DD: A new decision diagram for efficient problem solving in permutation space. In *14th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *LNCS*, pages 90–104. Springer, 2011.

[12] Akimitsu Ono and Shin-ichi Nakano. Constant time generation of linear extensions. In *15th International Symposium on Fundamentals of Computation Theory (FCT)*, volume 3623 of *LNCS*, pages 445–453. Springer, 2005.

[13] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithmics*, 11(1.7):1–24, 2006.

[14] Gara Pruesse and Frank Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, 1994.

[15] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

## Appendix: Proof of Lemma. 2

*Proof.* The proof is done inductively over $w$.

(Induction Basis)
When $w = 1$, $\sum_{k=1}^{1} l_k = n$ implies $l_1 = n$. Therefore,

$$\prod_{k=1}^{1}(l_k + 1) = l_1 + 1 = n + 1 = \left(\frac{n}{1} + 1\right)^1$$

holds. This directly proves the induction basis.

(Induction Step)
We suppose $\prod_{k=1}^{x}(l_k + 1) \leq \left(\frac{n}{x} + 1\right)^x$ holds when $\sum_{k=1}^{x} l_k = n$. We will prove that $\prod_{k=1}^{x+1}(l_k + 1) \leq \left(\frac{n}{x+1} + 1\right)^{x+1}$ holds when $\sum_{k=1}^{x+1} l_k = n$. First, we have

$$\prod_{k=1}^{x+1}(l_k + 1) = (l_{x+1} + 1) \cdot \prod_{k=1}^{x}(l_k + 1).$$

Here, $\sum_{k=1}^{x+1} l_k = n$ implies $\sum_{k=1}^{x} l_k = n - l_{x+1}$. Therefore, we have

$$\prod_{k=1}^{x+1}(l_k + 1) \leq (l_{x+1} + 1) \cdot \left(\frac{n - l_{x+1}}{x} + 1\right)^x = f(l_{x+1})$$

from the induction hypothesis. Let $a$ be $l_{x+1}$ to simplify. The first-order differentiation of $f(a)$ is calculated as follows.

$$\begin{aligned} f'(a) &= \left(\frac{n-a}{x} + 1\right)^x + (a+1) \cdot \left(-\frac{1}{x}\right) \cdot x\left(\frac{n-a}{x} + 1\right)^{x-1} \\ &= \left(\frac{n-a}{x} + 1 - a - 1\right) \cdot \left(\frac{n-a}{x} + 1\right)^{x-1} \\ &= \frac{n - (x+1)a}{x} \cdot \left(\frac{n-a}{x} + 1\right)^{x-1}. \end{aligned}$$

If $x = 1$, $f'(a) = \frac{n-(x+1)a}{x}$ and hence the maximum is $f(\frac{n}{x+1}) = (\frac{n}{x+1} + 1)^{x+1}$. Therefore, this satisfies the induction step. If $x \geq 2$, we obtain the extrema

$f(\frac{n}{x+1}) = (\frac{n}{x+1}+1)^{x+1}$ and $f(n+x) = 0$. The second-order differentiation of $f(a)$ is also calculated as follows.

$$\begin{aligned} f''(a) &= -\frac{x+1}{x}\cdot\left(\frac{n-a}{x}+1\right)^{x-1} + \frac{n-(x+1)a}{x}\cdot\left(-\frac{1}{x}\right)\cdot(x-1)\left(\frac{n-a}{x}+1\right)^{x-2} \\ &= \left\{-(x+1)\cdot\left(\frac{n-a}{x}+1\right) - (x-1)\cdot\frac{n-(x+1)a}{x}\right\}\cdot\frac{1}{x}\left(\frac{n-a}{x}+1\right)^{x-2} \\ &= \{(x+1)a - 2n - x - 1\}\cdot\frac{1}{x}\left(\frac{n-a}{x}+1\right)^{x-2}. \end{aligned}$$

Since $\frac{1}{x}(\frac{n-a}{x}+1)^{x-2}$ is always positive when $1 \le a \le n$ and $2 \le x$, $f(a)$ is upward-convex if $a < \frac{2n}{x+1}+1$, while $f(a)$ is downward-convex otherwise. Because $\frac{n}{x+1} < \frac{2n}{x+1}+1 \le n+x$ holds, we can conclude $f(\frac{n}{x+1}) = (\frac{n}{x+1}+1)^{x+1}$ is the maximum of $f(a)$ for all $1 \le a \le n$, and

$$\prod_{k=1}^{x+1}(l_k+1) \le (l_{x+1}+1)\cdot\left(\frac{n-l_{x+1}}{x}+1\right)^{x} = f(l_{x+1}) \le \left(\frac{n}{x+1}+1\right)^{x+1}$$

holds. This proves the induction step. □