Thesis submitted to obtain the title
Doctor of Information Science
平成28年度　博士学位論文

Studies on Permutation Set Manipulation Based on Decision Diagrams

決定グラフに基づく順列集合処理に関する研究

Yuma Inoue

井上 祐馬

Laboratory for Large-Scale Knowledge Processing,
Division of Computer Science and Information Technology,
Graduate School of Information Science and Technology
Hokkaido University

北海道大学 大学院情報科学研究科
情報理工学専攻 大規模知識処理研究室

February 2017

Abstract

In real life, we often face ordering problems of items: sorting items with some priorities, processing many tasks sequentially, ranking search results on the Web, traveling famous sites one by one, and so on. In mathematical terms, ordering of items is called a *permutation*. Since a permutation can be considered as a bijective function on a set, permutations can also represent e.g., reversible functions and pair-matching of items.

There are a lot of previous research results to find a "good" permutation for problems on which solutions can be described as permutations. For example, the *traveling salesman problem* requires us to find a "shortest" path consisting of a permutation of all vertices, and the *single-machine scheduling problem* requires us to find a permutation of all tasks with the "minimum" penalty. On the other hand, in real situations, conditions such as roads or priorities can be dynamically changed. In such cases, we have to calculate a "good" permutations again and again.

Enumeration of permutations is a way to overcome difficulties of dynamic changes. If we store all solutions and index to extract a "good" permutation from the solutions, we may obtain a "good" permutation faster than repeatedly calculating under new conditions. Furthermore, enumeration of all solutions is useful for other applications: obtain the number of solutions, random sampling of solutions, extracting solutions satisfying additional conditions, etc. For several conditions, enumeration of permutations has been also deeply studied. However, the number of the permutation is huge, namely factorial in the number of items. Hence the listing all permutations seems to be infeasible even for a few items.

An idea to avoid the factorial explosion is the usage of a compressed data structure to store and to index solutions. In this thesis, we focus on *permutation decision diagrams* (π*DDs*) as such a data structure. This data structure can store permutations compactly and possesses a permutation-set algebra including union and intersection. Moreover, manipulation and extraction of permutations in the data structure can be achieved in time depending only on the size of πDDs, not on the number of permutations represented by πDDs. This means that if the solutions could be well compressed by a πDD, manipulation of the solutions can also be efficiently processed.

Although a πDD is a powerful data structure, there are few results applying πDDs

to permutation problems. Hence, in the present thesis we purposely use πDDs for several permutation problems, and analyze their performance experimentally and theoretically. Chapter 3 provides results indicating that the usage of πDDs is effective for the following two applications: reversible circuit debugging and cycle-type partition of a permutation set. For these problems, it is worthless that we explicitly enumerate all the solutions and add them to a πDD because we cannot avoid calculation for a factorial number of permutations. We thus analyze the properties of the problems and propose πDD-construction algorithms for each problem based on the properties, utilizing πDD operations cleverly and/or reducing the number of πDD operations in the algorithms. In Section 3.1, we tackle *reversible circuit debugging problems*. Reversible circuits specify reversible functions, whose inputs are uniquely identified by their output. This property means reversible functions are permutations. For manufacture of reversible circuits, it is desirable to check whether a circuit specify an expected function or not. In previous work, a SAT-based algorithm and a πDD-based algorithm have been proposed for this purpose, they however cannot scale for circuits with many inputs and gates, and cannot identify error-responsible gates. We gives a theoretically faster algorithm to debugging circuit with a single error gate compared with previous methods. Although this algorithm does not use πDDs, the algorithm can be extended to a problem with multiple errors thanks to πDDs. This algorithm is the first algorithm to detect error positions for multiple errors model. In Section 3.2, *cycle-type partition* of permutation sets is investigated. The cycle-type of a permutation is a vector representing the distribution of *cycles* in the permutation. Two permutations are called cycle-type equivalent if they have the same cycle-type. The partition of a permutation set with respect to cycle-type equivalence plays an important role in combinatorics. We indicate that cycles of permutations have a property suitable to πDD representation, and provide a partition algorithm utilizing the advantage. Experimental evaluation confirms that this algorithm outperforms a naïve method and an existing πDD-based method.

On the other hand, we also face negative results for some applications, as shown in Chapter 4. This motivates us to invent a modified version of πDDs, so-called *Rot-πDDs*. We show that applying Rot-πDDs to such problems, which include enumeration of Eulerian trails, topological orders of directed graphs, and pattern-avoiding permutations, improves compression ratio and runtime of construction methods. In Section 4.2, we enumerate *Eulerian trails*, walks in an undirected graph such that each edge is used exactly once. Although listing algorithms have been proposed, they have never aimed to compress Eulerian trails and thus suffer from the time and space complexity to manipulate solutions on memory. We propose a Rot-πDD-construction algorithm based on a dynamic programming technique for counting, and show that the size of Rot-πDDs are theoretically bounded, and the analysis indicates that Rot-πDDs are preferable to πDDs. In Section 4.3, we enumerate *topological orders*, vertex orders on a directed

graph such that there is no edge from right to left in the order. For the topological ordering problem, enumeration and finding a solution under dynamic graph modifications have been individually studied. For enumeration, we show that the similar dynamic programming approach for Rot-πDD construction can also be applied to the topological ordering problem. Moreover, we propose a new Rot-πDD operation that realizes dynamic addition of edges. This implies Rot-πDDs can store all topological orders on a dynamic growing graph, and such an algorithm has never been proposed as far as the author know. In Section 4.4, we enumerate *pattern-avoiding permutations*, which are permutations whose subsequences do not have the same relative order as a pattern permutation. Some classes of pattern-avoiding permutations correspond to solutions of several other permutation problems. Thus permutation patterns are frequently used as the characterization of permutation problems. For enumeration of pattern-avoiding permutations, search-based algorithms have been proposed for general permutation patterns, and several specific algorithms have been proposed for restricted classes of patterns. We propose a πDD construction algorithm as a first step. This algorithm runs faster than the previous methods for general patterns. In addition, we show that, by using Rot-πDDs instead of πDDs, the sizes of Rot-πDDs in the middle of the algorithm are exactly smaller than the sizes of πDDs. This indicates that Rot-πDDs seem to be preferable to πDDs for the problem, and experimental results indeed confirm the prediction. It is a natural question how we choose preferable decision diagrams for each permutation problem. Section 4.5 includes suggestions for selection of decision diagrams with additional experimental data. We mainly investigate the effects of *disorder measures* for permutations, which evaluate hardness of sorting for the permutations from several aspects.

In Chapter 5, we attack a permutation problem for decision diagrams themselves: *variable ordering* of decision diagrams. Ordering of variables dramatically affects the size of a decision diagram, as we will see later. However, it is known that the variable ordering problem is NP-complete in general. We focus on decision diagrams to represent subgraphs of a given graph. We investigate that the size of decision diagrams can be bounded by a well-known graph parameter *path width*. Thus we propose a heuristic search algorithm optimizing the path width yielded by a variable order. Experimental results show that this algorithm generates orders with smaller path widths than the existing meta-heuristic approach, and hence may accelerate many decision diagram based algorithms.

Through a series of studies in this thesis, we develop several techniques to apply πDDs and Rot-πDDs to various permutation problems, and show the effectiveness of permutation decision diagrams by theoretical and experimental evaluations of the performance. Furthermore, we give commonly utilized ideas among similar problems and preliminary experiments for efficiency evaluations focusing on disorder measures of

permutations. These will be hints when we utilize πDDs and Rot-πDDs for other permutation problems. In addition, we provide a new method that improves a part of existing decision diagram based algorithms. It can be expected that the results in this thesis will lead to efficient methods for practical applications of permutation problems in this thesis and many other permutation problems.

Contents

Chapter 1

Introduction

1.1 Background

A *permutation* is a bijective function $\pi : A \to A$ from a set A to itself. Without loss of generality, we suppose A is a finite set of integers $\{1, \ldots, n\}$, then π can be considered as a numerical sequence $(\pi(1), \ldots, \pi(n))$. Since each of $\{1, \ldots, n\}$ appears exactly once in this sequence, a permutation can represent ordering of items.

In real life, we often face problems related to permutations. For example, we sometimes want to know short routes to visit all the places we should visit exactly once for a given road network. This is known as the *traveling salesman problem*, and we should compute appropriate permutations of places in the problem. For another instance, we sometimes want to process many tasks with deadlines sequentially as less deadline violations as possible. This is known as the *single-machine scheduling problem*, and we should compute appropriate permutations of tasks in the problem. We call problems whose solutions can be described by permutations *permutation problems*. There are many other permutation problems, briefly:

- Sorting: Rearrange books into a bookshelf in the order of their volumes.
- Ranking: Suggest search results on the result pages in the order according to somehow demands.
- Matching: Make n pairs each of which consists of a person from a group A and another person from a group B, where the number of persons of A and B is n.
- Encoding: Store strings in the form of secure/compressed codes. One-to-one correspondence ensures decoding ability.

Thus, permutations play a important role in combinatorial problems related to several real-life applications, and it is desired to efficiently solve such permutation problems.

Many researches investigate different types of objectives for individual permutation problems. As an instance of permutation problems, we focus on the *topological sort problem* here, which will be also discussed later in Section 4.3. In the topological sort problem, for given a directed graph, we aim to compute a *topological order*, which is a vertex order of a directed graph such that there is no directed edge contradicting the vertex order, i.e., if an edge from v_i to v_j, then $i < j$ must hold in the vertex order $(v_1, \ldots, v_n)$. It corresponds to a scheduling problem of tasks with relative priorities. Several types of objectives for the topological sort problem have been investigated, for example:

- Finding a single (optimal) solution [44, 78],
- Sampling a uniformly random solution [18],
- Enumerating all solutions [70, 64],
- Counting the number of solutions [53], and
- Updating a solution under dynamic graph modifications [8, 65].

Since application problems require different types of objectives, researchers have studied algorithms to obtain each type of solutions individually, as described above.

On the other hand, if we store a set of all the permutations, we can immediately obtain a permutation, a uniformly random permutation, all the permutations, and the number of solutions from the set. We also can find an optimal permutation from the set by evaluating each permutation and handle dynamic modification of the set by applying modifications to each permutation or adding/removing permutations. Therefore, storing all permutations has a comprehensive potential for several objectives of each permutation problem. However, the number of permutations on the set $\{1, \ldots, n\}$ is $n!$, very huge to store them in a naïve way. In addition, it is inefficient to evaluate and apply modifications to all the permutations because it requires time linear in the cardinality of a solution set. Thus we aim to compress a set of permutations and to index them for manipulation of permutations in the set without restoring and traversing them. This is our target problem in the present thesis: for permutation problems, we store all the solutions of the problem in a compact and indexed form to efficiently manipulate them on memory, such as random sampling, counting the cardinality, and dynamic modification.

1.2 Related Work

Many researches investigated compression of a single permutation. An important fact to study compression of a permutation is that naïve array representation of permutations

on integers is already an asymptotically optimal encoding from a point of view of information theory; to distinguish $n!$ permutations, we need $\lceil \log(n!) \rceil \simeq n \log n - 1.44n$ bits[1] for each permutation, and naïve array representation stores n integers encoded in $\lceil \log n \rceil$ bits. This means that the naïve array representation has the redundancy of $n\lceil \log n \rceil - \lceil \log(n!) \rceil \simeq n(\lceil \log n \rceil - \log n) + 1.44n \leq 2.44n = \Theta(n)$ bits only, compared with the theoretical lower bound.

One of research directions of compression of a single permutation is to reduce the redundancy of $\Theta(n)$ term of the naïve array representation. In this context, Munro et al. [62] have proposed a succinct data structure for permutations with $\lceil \log(n!) \rceil + O(n(\log\log n)^5/(\log n)^2)$ bits representation, which supports $\pi()$ and $\pi^{-1}()$ queries in $O(\log n/\log\log n)$ time. This means that the succinct data structure has only $o(n)$ redundancy for function calculation queries, and thus improves the redundancy of the naïve array representation.

Another direction of single-permutation compression is *adaptive* compression: the smaller disorderedness a given permutation has, the shorter encoding the compression scheme assigns to the permutation. Barbay and Navarro [5] have proposed a data structure with at most $n + n\log \mathit{Runs}(\pi)$ bits, which supports $\pi()$ and $\pi^{-1}()$ in $O(\log \mathit{Runs}(\pi)/\log\log n)$ time for a given permutation π, where $\mathit{Runs}(\pi)$ is the number of runs in π, which are the positions i of descents $\pi(i) > \pi(i+1)$.

Unfortunately, the above two directions are for a single permutation. These approaches cannot avoid factorial explosion for a large permutation set, because it requires the size linear in the cardinality of a permutation set. On the other hand, the lower bound of the size of an encoded permutation set is also factorial: $\lceil \log 2^{n!} \rceil = n!$ bits are required. Thus, we would need to introduce a similar idea to adaptive compression: we assign short encodings to permutation sets that frequently arise in application problems.

For permutation groups, there is a well-known compact representation by a *strong generating set*. Sims [75] has proposed the first algorithm, called *Schreier-Sims algorithm*, to compute a strong generating set for given generators of a permutation group, and Knuth [49] gives the complexity analysis: $O(n^5 + n^2 m)$ time and $O(n^3 \log n)$ space, where m is the number of given generators. This representation can process several queries, e.g. membership query, and thus it is implemented in several computer algebra systems, e.g. GAP [34]. However, this representation can be used for only permutation groups and thus cannot be used for arbitrary permutation sets directly, because it represents a permutation group as the set of its generators, whereas an arbitrary permutation set cannot be generated by a generator set in general.

Permutation decision diagrams (π*DDs*) [60] are data structure for arbitrary permutation sets derived from *Zero-suppressed Binary Decision Diagrams* (*ZDDs*) [59], which

[1] in this thesis, we use a notation log to represent the logarithm to the base 2.

are used to represent a set of combinations. The key idea of πDDs is *transposition decomposition* of permutations, the sequence of exchanges (transpositions) to make the identity permutation to a given permutation. A permutation set can be represented by a decision tree with transposition nodes: a path to a true-node (resp. a false-node) represents the transposition decomposition of a permutation in the set (resp. not in the set). A πDD is a decision tree compressed by sharing and deleting nodes.

The precise upper bound of the size of a πDD is not revealed: although a rough upper bound is 2^{n^2}, in several practical cases, a πDD can store permutations compactly. πDDs also support permutation-set algebra such as union and intersection between permutation sets. Moreover, manipulation and extraction of permutations in the data structure can be achieved by recursive procedures that run in time depending only on the size of πDDs, not on the number of permutations represented by πDDs. This means that if the solutions can be well compressed as a πDD, manipulation of the solutions would also be efficiently processed.

Although a πDD is a powerful data structure, there are few results applying πDDs to permutation problems, e.g. counting primitive sorting networks [46]. Thus many open problems about πDDs remain as follows:

- Applications for several permutation problems should be examined to evaluate the practical performance (e.g., compression ratio and runtime for construction and queries) of πDDs.
- Are there problems for which the usage of πDDs is very effective or not? If so, what kind of properties of such problems affects the performance of πDDs? Can we categorize such problems by mathematical characterization and use the same (or similar) techniques with πDDs to efficiently solve problems in the same category?
- For problems unfavorable for πDDs, can we improve the performance of πDDs by modifying πDDs?
- Can we theoretically evaluate the performance of πDDs? This seems to be challenging work because the theoretical-size analysis of ZDDs is known to be difficult in general.

We tackle the above problems in this thesis. We hence purposely use πDDs for several permutation problems, and analyze their performance experimentally and theoretically.

1.3 Contributions

Contributions in this thesis are summarized as follows:

1. We apply πDDs to the following two permutation problems:

   - Debugging erroneous reversible circuits: A *Reversible circuit* specifies a reversible function, whose inputs and outputs have one-to-one correspondence, i.e., a permutation. Incidental errors in manufacture may make a reversible circuit work as an unexpected function. We use πDDs for an algorithm to check whether a reversible circuit has errors or not, and if so, detect the error positions and debug them. This is a first algorithm to debug erroneous circuits with multiple errors.
   - Cycle-type partition of a permutation set: Permutations can be decomposed into *cycles*, and the *cycle-type* of a permutation is the distribution of the length of cycles. *Cycle-type partition* of a permutation set can be obtained by the equivalence relation with respect to cycle-types. We provide an algorithm such that for given a πDD representing an input permutation set, it returns a set of πDDs each of which corresponds to a permutation set in the partition. For this problem, there is an existing work utilizing πDDs [87]. We indicate a good relation between cycle decomposition and πDD structure, and the proposed method utilizing this property outperforms the existing method in computational experiments.

2. We propose a new variation of πDDs, *Rotation-based πDDs* (*Rot-πDDs*). We also apply Rot-πDDs to the following three permutation problems:

   - Enumeration of Eulerian trails: *Eulerian trails* are traversals of a graph in which each edge of the graph is passed exactly once, i.e., a permutation of edges. We utilize a dynamic programming approach to count the number of all Eulerian trails for construction of a Rot-πDD representing all Eulerian trails. Complexity analysis of the construction algorithm indicates that computation time and the size of a resulting Rot-πDD are theoretically bounded, and Rot-πDDs are preferable to πDDs for Eulerian trails.
   - Enumeration of topological orders: Topological orders, which are briefly introduced in Section 1.1, are also studied in this thesis. We show that a dynamic programing based construction can be utilized for the topological sort problem, and the time and space complexity of the algorithm are well-bounded by a graph parameter, *minimum path cover*. We also provide a Rot-πDD operation for extracting permutations in which an element a precedes another element b. This corresponds to processing dynamic addition of an edge to a directed graph. Experimental results confirm the efficient compression of Rot-πDDs: 3.7×10^{41} topological orders can be represented by a Rot-πDD with the size 2.2×10^{7}.

- Enumeration of pattern-avoiding permutations: A text permutation τ *contains* a pattern permutation π if there are subsequences of τ that have the same relative order to π. Otherwise, τ *avoids* π. We propose enumeration algorithms for pattern-avoiding permutations using πDDs and Rot-πDDs. We show that the sizes of Rot-πDDs in the middle of the algorithm are smaller than ones of πDDs. Experiments demonstrate that decision diagram based algorithms perform faster than a naïve method and the existing method used in a software for permutation pattern analysis, and confirm the Rot-πDD based algorithm is superior to the πDD-base algorithm in terms of both of construction time and the size of decision diagrams.

3. We investigate what parameters of permutations affect compression ratio of decision diagrams. We focus on *disorder measures* as parameters of permutations, which are measures for hardness to sort a permutation. We give some suggestions to properly use decision diagrams for each problem based on the contributions for the above permutation problems and preliminary experiments for disorder measures.

4. We tackle to ordering problem in decision diagrams, *variable ordering*. As we will see in the following chapters, an order of variables dramatically affects the compression performance of decision diagrams. On the other hand, computation of a variable order yielding the minimum decision diagram is known to be NP-complete in general [11]. In this thesis, we focus on ZDDs representing subgraphs of a given graph and *frontier-based search*, a ZDD construction algorithm for subgraphs. We indicate the relation between frontier-based search and a graph parameter *path width* of a given graph. We provide a meta-heuristic algorithm utilizing this relation for computation of a good variable order. In computational experiments, the proposed algorithm yields better orders compared with variable orders calculated by the algorithms used in a frontier-based search library.

1.4 Thesis Organization

In the following Chapter 2, we introduce definitions and notations about permutations, graphs, and decision diagrams, which are our objectives and tools used in this thesis. In Chapter 3, we provide the two instances of applications of πDDs: reversible circuit debugging and cycle-type partition of a permutation set. These results show that appropriate πDD usage makes algorithms faster and less space, or gives new ideas to enumerate permutations. Chapter 4 introduces a new variation of πDDs, Rot-πDDs, invented by the author. Chapter 4 also includes the three applications of Rot-πDDs:

enumeration of Eulerian trails, topological orders, and pattern-avoiding permutations. Furthermore, we investigate proper choice of πDDs and Rot-πDDs for permutation problems in the last of Chapter 4, focusing on disorder measures of permutations. In Chapter 5, we introduce a permutation problem for decision diagram construction: variable ordering. We give an algorithm to find a good variable order, namely a permutation of variables, in terms of yielding smaller decision diagrams. Chapter 6 summarizes the results in this thesis and indicates future work.

Chapter 2

Preliminaries

In this chapter, we introduce required definitions and notations to describe the contributions in this thesis. More precisely, we first introduce permutations, which are the main objectives of this thesis. Next, we introduce graphs since our data structure "decision diagrams" are in the form of directed graphs, and they also arise on our application problems frequently. Finally, we introduce decision diagrams, which are our main tools to solve permutation problems efficiently.

2.1 Permutations

We first provide basic definitions and notations of permutations. We denote the set $\{1,\ldots,n\}$ by $[n]$.

Definition 2.1.1. *A* permutation *is a bijective function mapping a set onto itself. Without loss of generality, a permutation on a finite set with cardinality n can be considered as a function $\pi : [n] \to [n]$, and we call it a permutation of length n, or an n-permutation shortly. An n-permutation π can be written in the one-line form as $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, and we denote $\pi_i = \pi(i)$. Thus, an n-permutation can be considered as a numerical sequence of length n such that each of integers $1,\ldots,n$ appears exactly once in the sequence.*

Example 2.1.1. *A numerical sequence $\pi = (4,3,1,2)$ represents a 4-permutation, and $\pi_3 = 1$.*

Since a permutation is a function, we can define composition of two permutations.

Definition 2.1.2. *The composition $\pi \cdot \sigma$ of two permutations π and σ is $\pi \cdot \sigma = (\sigma(\pi(1)), \ldots, \sigma(\pi(n)) = (\sigma_{\pi_1}, \ldots, \sigma_{\pi_n})$. Note that we apply the leftmost permutation first. We sometimes omit $\cdot$ when it is obvious from the context.*

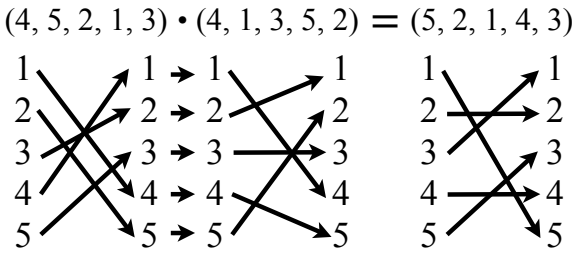


Figure 2.1. An example of composition of two permutations

Example 2.1.2. *Let* $x = (4,5,2,1,3)$ *and* $y = (4,1,3,5,2)$*, then* $x \cdot y = (5,2,1,4,3)$ *(see Figure 2.1 for visualization).*

We notice that composition of permutations is non-commutative.

We also define some special permutations.

Definition 2.1.3. *We say that a permutation* π fixes *i if* $\pi_i = i$*. An n-permutation* π *is an* identity permutation *if* π *fixes all the elements* $1,\ldots,n$*. As a numerical sequence, the identity permutation of length n is the increasing sequence* $(1,2,\ldots,n)$*. We denote an identity permutation by* ι*.*

Definition 2.1.4. *The inverse* π^{-1} *of a permutation* π *is the inverse function of* π*, i.e.,* $\pi^{-1}(\pi(i)) = i$ *for all* $1 \le i \le n$*. Hence,* $\pi \cdot \pi^{-1} = \pi^{-1} \cdot \pi = \iota$ *holds.*

Definition 2.1.5. *A transposition* $\tau_{i,j}$ $(i < j)$ *is a permutation fixes all the elements except i and j, and* $\tau_{i,j}(i) = j$ *and* $\tau_{i,j}(j) = i$*. Namely,* $\tau_{i,j} = (1,\ldots,i-1,j,i+1,\ldots,j-1,i,j+1,\ldots n)$.

We can decompose a permutation into a sequence of transpositions [60].

Definition 2.1.6. *Transposition decomposition of a permutation* π *is a sequence of transpositions recursively computed as follows: If* π *is an identity permutation, we return an empty sequence. Otherwise, let x be the maximum unfixed element. Then* $\pi' = \pi \cdot \tau_{x,\pi_x}$ *is recursively decomposed and compose* τ_{x,π_x} *to the right of the obtained composition. This is correct since* $\pi' \cdot \tau_{x,\pi_x} = \pi \cdot \tau_{x,\pi_x} \cdot \tau_{x,\pi_x} = \pi$*. This procedure will terminate because* $\pi' \cdot \tau_{x,\pi_x}$ *fixes x and hence the maximum unfixed element properly decreases in the recursive procedure.*

Proposition 2.1.1. *Any n-permutation can be uniquely represented as a transposition decomposition with at most* $n-1$ *transpositions.*

Example 2.1.3. *We demonstrate the transposition decomposition of a permutation* $\pi = (5,4,2,1,3)$ *(see Figure 2.2 for visualization). The maximum unfixed element of* π *is* 5*, and the* 5*-th element* π_5 *is* 3*, hence we exchange* 5 *and* 3*, and obtain* $\pi' = \pi \cdot \tau_{3,5} =$

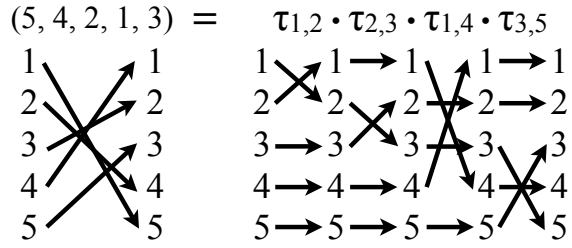


Figure 2.2. An example of transposition decomposition of a permutation

$(3,4,2,1,5)$. *Since the maximum unfixed element of* π' *is* 4 *and the* 4*-th element of* π' *is* 1*, we then obtain* $\pi'' = \pi' \cdot \tau_{1,4} = (3,1,2,4,5)$. *Repeating this procedure, we finally obtain* $\pi = (5,4,2,1,3) = \tau_{1,2} \cdot \tau_{2,3} \cdot \tau_{1,4} \cdot \tau_{3,5}$.

Permutations can be decomposed by another fashion: *cycle decomposition*.

Definition 2.1.7. *A* cycle $(a_1 \ \ldots \ a_k)$*, where* $a_i \in \{1,2,...,n\}$ *and* $a_i \neq a_j$ *for* $i \neq j$*, denotes a permutation* π *such that* $\pi(a_i) = a_{i+1}$ *for* $1 \leq i < k$ *and* $\pi(a_k) = a_1$*, and* π *fixes the other elements. Here, k is called the length of a cycle, and* $(a_1 \ \ldots \ a_k)$ *is called a k-cycle.*

Definition 2.1.8. *Any permutation can be uniquely decomposed into composition of disjoint cycles. The* cycle decomposition *of a permutation is a sequence of disjoint cycles such that composition of the cycles is the permutation.*

Although the order of elements in cycles and the order of compositions are arbitrary, we define the *normal form* of cycle decomposition as follows. First, we cyclically shift each cycle such that the minimum element in the cycle is at the first position. Next, we reorder cycles in increasing order of the first elements.

Example 2.1.4. *Since* $(4,6,1,3,5,2) = (5)(6\ 2)(4\ 3\ 1)$*, a permutation* $(4,6,1,3,5,2)$ *is decomposed into three disjoint cycles:* (5)*,* $(2\ 6)$*, and* $(1\ 4\ 3)$*. This can be confirmed by the graph visualization in Figure 2.3. Its normal form is* $(1\ 4\ 3)(2\ 6)(5)$*. For instance,* $(1\ 4\ 3)$ *is a* 3*-cycle.*

We denote the set of all the n-permutations by S_n. Unless otherwise noted, we hereafter use Greek alphabets for a permutation (e.g. π), and uppercase English letters for a permutation set (e.g. P) in the context of permutations.

2.2 Graphs

We introduce basic definitions and notations for undirected graphs.

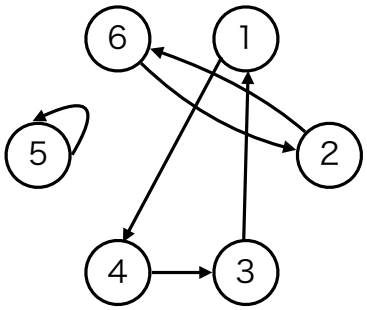
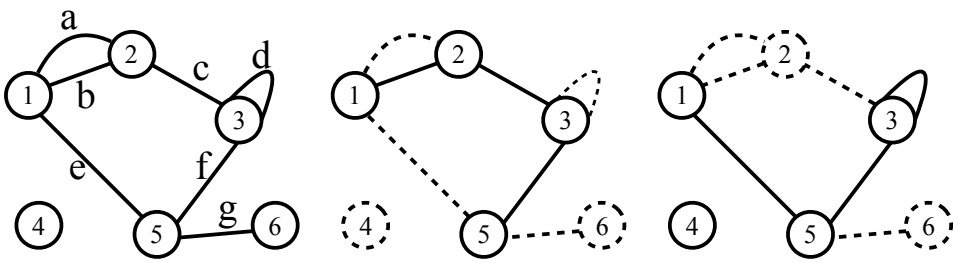


Figure 2.3. Graph representation of the cycle decomposition of a permutation $(4,6,1,3,5,2)$



(a) a graph G (b) a subgraph of G (c) an induced subgraph of G

Figure 2.4. A graph, its subgraph, and its induced subgraph

Definition 2.2.1. *An* undirected graph *G consists of two sets V and E, where V is a* vertex *set and E is an* edge *multiset. An edge is unordered pairs of vertices in V. We denotes the cardinality $|V|$ of V by n and the cardinality $|E|$ of E by m. Without loss of generality, we assume that $V = [n]$.*

Example 2.2.1. *Figure 2.4(a) illustrates an undirected graph G with $n = 6$ vertices and $m = 7$ edges.*

Unless otherwise noted, we call an undirected graph as a graph for short.

Definition 2.2.2. *An edge $e = \{u, v\}$ is called a* self-loop *if $u = v$. Two different edges e and f are said to be* multiple edges *if $e = f$. A graph is* simple *if the graph has no self-loop and no multiple edge.*

Example 2.2.2. *A graph G in Figure 2.4(a) is not simple because it has a self-loop $d = \{3,3\}$ and multiple edges a and b.*

Next, we introduce some concepts about graphs that will be used later.

Definition 2.2.3. *Two vertices u and v is* adjacent *if an edge $\{u,v\}$ is in E. For edge $e=\{u,v\}$, u and v are called* endpoints *of e, and e is* incident *to u and v. The degree $d(v)$ of a vertex $v \in V$ of $G=(V,E)$ is the number of edges incident to v, here each self-loop is counted twice. Vertices with degree 0 are called to be* isolated. Neighbors *of a vertex v are a set of vertices adjacent to v, and denoted by $N(v)$. Neighbors of a vertex set S are similarly defined as $N(S)=\sum_{v\in S}N(v)\setminus S$.*

Example 2.2.3. *We refer the graph G in Figure 2.4(a). The degree of the vertex 5 in G is 3, and the degree of the vertex 3 is 4. The vertex 4 is an isolated vertex. The neighbors of the vertex 2 is $\{1,3\}$, and the neighbors of $\{2,5\}$ is $\{1,3,6\}$.*

Definition 2.2.4. *A* walk *on a graph is an alternating sequence $(v_1,e_1,v_2,e_2,\ldots,v_k,e_k,v_{k+1})$ of vertices and edges such that $e_i=\{v_i,v_{i+1}\}$. A* trail *is a walk on which the same edge appears at most once. A* path *is a walk on which the same vertex appears at most once. A* cycle *is a closed path, i.e., $v_1=v_{k+1}$. The length of a walk (resp. trail, path, and cycle) is defined by the number k of edges in a walk (resp. trail, path, and cycle). When the vertices in a walk (resp. trail, path, and cycle) is obvious from the context, we use an edge sequence $(e_1,e_2,\ldots,e_k)$ to represent a walk (resp. trail, path, and cycle).*

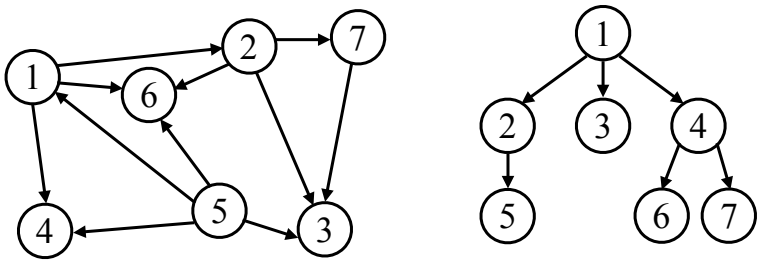
Example 2.2.4. *In the graph G in Figure 2.4(a), a sequence $(6,g,5,f,3,d,3,f,5,e,1)$ is a walk with length 5, but not a trail because edge f is appears twice. A sequence $(5,f,3,c,2,b,1,a,1)$ is a trail with length 4, but not a path because vertex 1 is appears twice. A sequence $(6,g,5,e,1,a,2,c,3)$ is a path with length 4. A sequence $(1,b,2,c,3,f,5,e,1)$ is a cycle with length 4.*

Definition 2.2.5. *A graph $G'=(V',E')$ is a* subgraph *of a graph $G=(V,E)$ if $V'\subseteq V$ and $E'\subseteq E$. We denote a subgraph* induced *by S as $G[S]=(S,E(S))$, where $E(S)$ is a multiset of all the edges in E incident to at least one vertex in S. We call $G[S]$ an* induced subgraph *of G.*

Example 2.2.5. *The graphs of Figure 2.4(b) and (c) are subgraphs of G in Figure 2.4(a). The graph of Figure 2.4(c) is also an induced subgraph $G[\{1,3,4,5\}]$ of G, whereas the graph of Figure 2.4(b) is not because, for example, vertices 1 and 2 are in the graph but edge a is not used.*

Definition 2.2.6. *Two vertices u and v are* connected *if there is a path from u to v. A graph G is connected if any two vertices in G are connected. Connected components of a graph is maximal connected subgraphs, i.e., connected induced subgraphs $G[S]$ such that for all $v\in V\setminus S$, $G[S\cup\{v\}]$ is not connected.*

Example 2.2.6. *In the graph G in Figure 2.4(a), vertices 2 and 6 are connected while vertices 3 and 4 are not connected. The connected components in G are $G[\{1,2,3,5,6\}]$ and $G[\{4\}]$.*



(a) a directed graph (b) a rooted tree

Figure 2.5. An example of directed graph

We also introduce directed graphs.

Definition 2.2.7. *A directed graph G is an ordered pair of two sets V and E, where V is a vertex set and E is an edge multiset. Each edge in a directed graph is an "ordered" pair of vertices in V. Thus, we consider that directed edges (u,v) and (v,u) are different, while undirected edges $\{u,v\}$ and $\{v,u\}$ are the same.*

Definition 2.2.8. Out-going edges *of a vertex $v \in V$ is the edges $\{e \mid e = (v,x) \in E, x \in V\}$.* In-coming edges *of a vertex $v \in V$ is the edges $\{e \mid e = (x,v) \in E, x \in V\}$.* Out-degree $d_-(v)$ *of a vertex v is the number of out-going edges of v.* In-degree $d_+(v)$ *of a vertex v is the number of in-coming edges of v.*

Example 2.2.7. *Figure 2.5(a) shows an example of a directed graph. The out-degree of the vertex* 2 *is 3, and the in-degree of the vertex* 2 *is 1.*

Definition 2.2.9. *A walk on a directed graph is an alternating sequence $(v_1, e_1, v_2, e_2, \ldots, v_k, e_k, v_{k+1})$ of vertices and edges such that $e_i = (v_i, v_{i+1})$. Trails, paths, and cycles on a directed graph are similarly defined ones on an undirected graph.* Directed acyclic graphs *(*DAG*) are directed graphs having no cycles.*

Example 2.2.8. *The directed graph in Figure 2.5(a) has no cycle. Hence the directed graph is a DAG.*

Definition 2.2.10. *A* rooted tree *is a DAG T with a root vertex r such that for all vertex v in T, there is only one path from r to v. We sometimes denote vertices in rooted trees by* nodes*. A node v is at the i-th level if the length of the path from r to v is $i-1$. A node with at least one out-going edge is called an* internal node*, whereas a node with no out-going edge is called a* leaf*. A node u pointed by an out-going edge from a node v is called a* child node *of v. Then, v is a parent of u.*

Example 2.2.9. *Figure 2.5(b) illustrates a rooted tree. At the* 2*-th level, there are three nodes* 2*,* 3*, and* 4*. Nodes* 1*,* 2*,* 3*, and* 4 *are internal nodes, and nodes* 5*,* 6*, and* 7 *are leaves. A node* 6 *is a child node of a node* 4*.*

x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

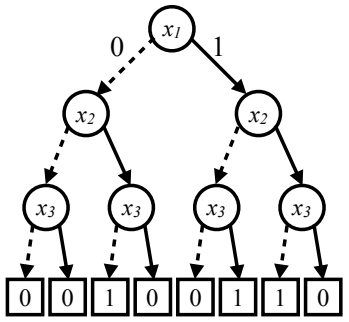


Figure 2.6. The truth table and the binary decision tree representing a logic function $(x_1 \vee x_2) \wedge (x_2 \oplus x_3)$

2.3 Decision Diagrams

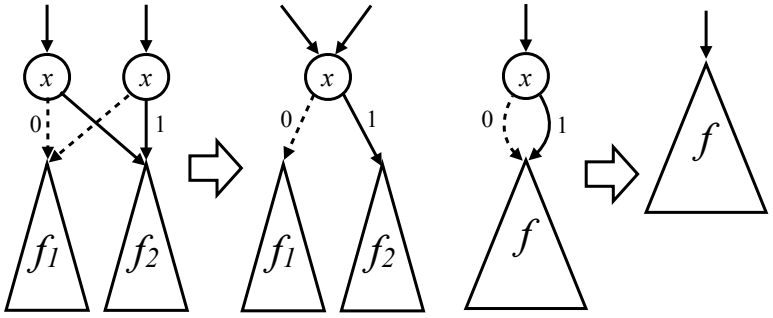
Permutation decision diagrams (PiDDs, or πDDs) [60] are data structures which canonically represent and efficiently manipulate a set of permutations. The structure of πDDs is based on *Zero-suppressed Binary Decision Diagrams* (*ZDDs*) [59], which are decision diagrams for sets of combinations (families of sets). In this section, we briefly review some decision diagrams related to contributions in this thesis.

2.3.1 Binary Decision Diagrams (BDDs)

In order to represent a logic function $f : \{0,1\}^n \to \{0,1\}$, we can use a *binary decision tree*, which is a rooted tree with nodes labeled by variables. To construct a binary decision tree, we first fix a variable order: the i-th variable appears only at the i-th level of a decision tree. Each internal node at the i-th level is labeled with the i-th variable x_i, and has exactly two out-going edges: a 0-edge and a 1-edge. Each leaf is labeled with 0 or 1. Each path from a root to a leaf corresponding to an assignment of variables; if a 1-edge from a node labeled with x_i is in a path, x_i is set to 1 in the corresponding assignment, whereas a 0-edge means that x_i is set to 0. If the path reaches a leaf, for the assignment corresponding to the path, f returns the label of the leaf, 0 or 1. Figure 2.6 provides an example of a logic function and its binary decision tree.

Binary decision Diagrams (*BDD*) [16] are compressed binary decision trees. The following two rules reduce a binary decision tree to a BDD, as visualized in Figure 2.7:

- sharing rule: equivalent nodes, which have the same labels and the same child nodes, are unified to one node and all 0-/1-edges pointing to these nodes re-point to the unified node.
- deleting rule: delete all nodes of which the 0-/1-edges point to the same node.



(a) sharing rule (b) deleting rule

Figure 2.7. Two reduction rules for BDDs

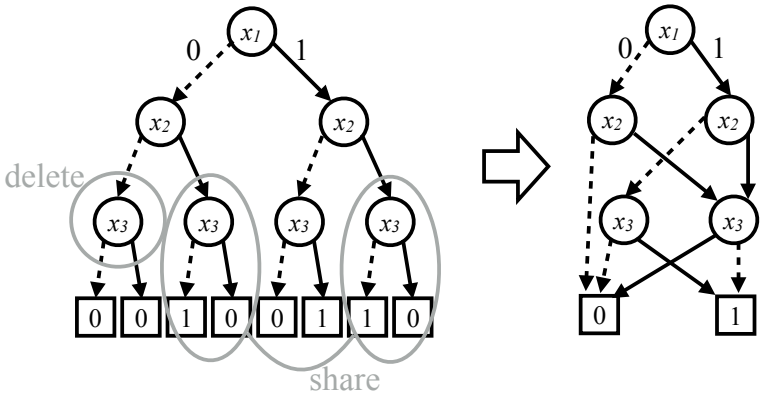


Figure 2.8. A binary decision tree and its reduced BDD

Figure 2.8 shows an example of a reduced BDD representing the logic function in Figure 2.6. This reduction can be achieved in time linear in the size of an original binary decision tree (or a non-reduced BDD) by, e.g., Algorithm R in [50]. We notice that the rule must reduce all leaves labeled with 0 (resp. 1) into a leaf labeled with 0 (resp. 1). We call the unified leave as the 0-*sink* and the 1-*sink*, respectively.

We also notice that a BDD obtained by repetition of applying the two rules whenever possible is canonical: A BDD has one-to-one correspondence to a logic function. Hence changing variable orders is only a way to affect the BDD structure and the compression ratio. We define the *size* of a BDD as the number of nodes in the BDD. Variable ordering of a BDD dramatically affects the size of BDDs for some logic functions. This indicates an appropriate variable order may achieve exponential improvement of compression. Unfortunately, Bryant [17] has proved that there is a function for which any variable orders yield exponentially large BDDs. In addition, the problem determining the best variable order, i.e. the order yields a BDD with the minimum size, is NP-complete [11]. However, in practice, many logic functions can be represented by a

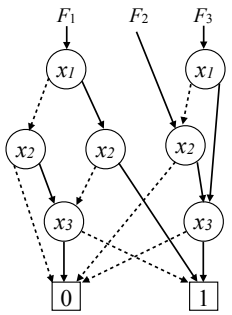


Figure 2.9. Three BDDs for F_1, F_2, and F_3 share a part of sub-BDDs.

small BDD in heuristically-detemined orders. We will deal with the variable ordering problem in Chapter 5 more deeply.

We also notice that a BDD is a DAG, not a tree, due to the reduction rules. Here, a *sub-BDD*, namely a subgraph consisting of a BDD node and its reachable nodes, is also a BDD. For example, in Figure 2.8, the sub-BDD pointed by the 1-edge from the root node (labeled with x_1) can be considered as a BDD for a logic function $f(x_2, x_3) = x_2 \oplus x_3$. This sub-BDD property is important in the following points of view:

- Shared BDD: two or more BDDs can share its sub-BDDs. Thus the nodes generated to construct multiple BDDs can be less than the sum of the size of the BDDs (see Figure 2.9 for example).

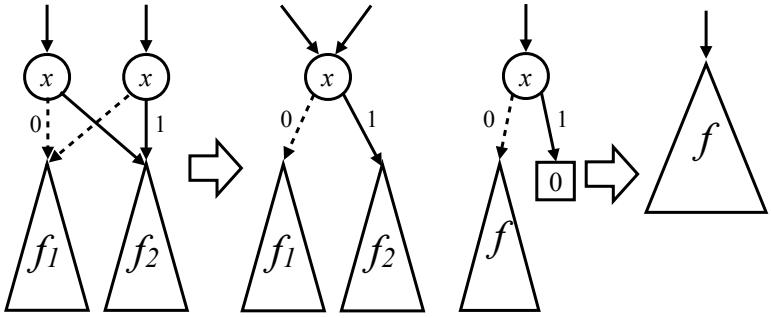
- Binary operations: logical binary operations such as AND, OR, and XOR between two BDDs can be calculated without explicitly restoring their truth tables. Briefly, recursive procedures compute binary operations between two BDDs, whose complexity is the product of the sizes of two BDDs.

Since the complexity of many operations on BDDs depends on the size of BDDs, rather than on the number of literals and clauses, we can also consider BDDs as data structure accelerating calculation on logic functions.

Hereafter, we refer a BDD B as a tuple (x, B_l, B_r) of three elements: the label x of the root node of B, the sub-BDD B_l pointed by the 0-edge from the root node, and the sub-BDD B_r pointed by the 1-edge from the root node. We can uniquely determine a BDD from this notation since two BDDs represented by the same notation should be unified into a single BDD according to the sharing rule.

2.3.2 Zero-suppressed Binary Decision Diagrams (ZDDs)

A *zero-suppressed binary decision diagram* (*ZDD*) is a modified version of BDDs to represent *combinatorial set* more efficiently. A BDD corresponds to a logic function,



(a) sharing rule (b) deleting rule

Figure 2.10. Two reduction rules for ZDDs, where (a) is the same as the BDD's sharing rule

and a logic function can be considered to represent a set of combinations as follows: For an assignment x for a logic function f, the variables x_i with $x_i = 1$ in the assignment compose a subset of variables. If $f(x) = 1$, then the combination $\{x_i : x_i = 1\}$ is in the set corresponding to f, whereas assignment x such that $f(x) = 0$ is not in the set.

Example 2.3.1. *A 4-arguments logic function f in Figure 2.6 corresponds to the set of combinations* $\{\{x_1,x_2\},\{x_1,x_3\},\{x_2\}\}$.

A ZDD is derived from a binary decision tree like a BDD. To obtain ZDDs, we use the same sharing rule as BDDs. On the other hand, the different deleting rule is used (as shown in Figure 2.10):

- deleting rule: delete all nodes of which the 1-edge points to the 0-sink.

Figure 2.11 shows the ZDD for the combination set $\{\{x_1,x_2\},\{x_1,x_3\},\{x_2\}\}$. Compared with BDDs, the deleting rule of ZDDs intends to compress sparse combinations more efficiently, since elements that do not appear in combinations are deleted. An example in Figure 2.11 indicates that the size of the ZDD is smaller than the size of the BDD for a sparse combinatorial set.

2.3.3 Permutation Decision Diagrams (πDDs)

We introduce πDDs, which represent sets of permutations, by utilizing the structure of ZDDs. As defined in Definition 2.1.6, any permutation can be uniquely decomposed into composition of transpositions. Hence, by assigning transpositions to nodes in a ZDD, each path in the ZDD represents a permutation. This is the basic idea of πDDs.

We introduce the order of transpositions $<$ so that $\tau_{x_1,y_1} < \tau_{x_2,y_2}$ if $y_1 > y_2$ holds, or $y_1 = y_2$ and $x_1 < x_2$ holds. We use this order as the fixed order in a πDD. In πDDs, the 0-sink represents an empty set and the 1-sink represents a singleton of the identity

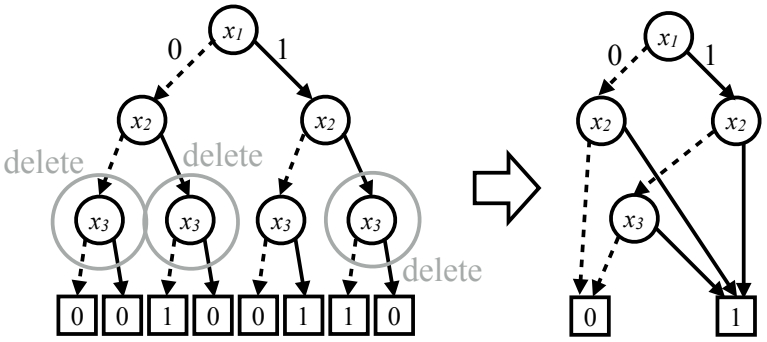


Figure 2.11. A binary decision tree and its reduced ZDD

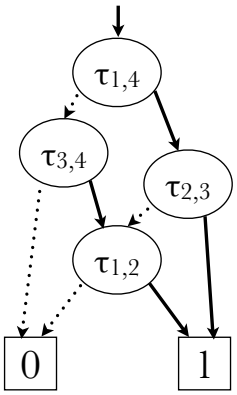


Figure 2.12. The πDD for $\{(2,1,4,3),(2,4,3,1),(4,3,2,1)\} = \{\tau_{1,2}\cdot\tau_{3,4}, \tau_{1,2}\cdot\tau_{1,4}, \tau_{2,3}\cdot\tau_{1,4}\}$.

permutation ι. We obtain compact and canonical πDDs by applying the two reduction rules in the same way to ZDDs. Figure 2.12 shows an example of a πDD.

Unless otherwise noted, we hereafter use calligraphic letters for decision diagrams (e.g. $\mathbb{P}$ for a permutation set P). We also refer a ZDD and a πDD as a tuple (x, Z_l, Z_r) of a label and two sub-DDs like BDDs.

Set operations on πDDs are inherited from ZDDs. Table 2.1 shows the πDD operations proposed in [60], which will be used in this thesis. While intersection, union, and set difference operations are available like ZDDs, the swap and Cartesian product operations are unique to πDDs. Here the Cartesian product of πDDs differs from the usual Cartesian product of sets: the set of compositions for all pairs of permutations in one permutation set and those in the other permutation set. In particular, the Cartesian product is very useful because compositions over permutations results in rearrangements. That is, by applying the Cartesian product operator, we can simultaneously execute rearrangements of multiple numerical sequences.

Table 2.1. πDD operations on two πDDs P and Q.

$\mathbb{P} \cap \mathbb{Q}$	return intersection $\{\pi \mid \pi \in P \text{ and } \pi \in Q\}$.
$\mathbb{P} \cup \mathbb{Q}$	return union $\{\pi \mid \pi \in P \text{ or } \pi \in Q\}$.
$\mathbb{P} \setminus \mathbb{Q}$	return difference $\{\pi \mid \pi \in P \text{ and } \pi \notin Q\}$.
$\mathbb{P}.Swap(x,y)$	return swapped permutations $\{\pi \cdot \tau_{x,y} \mid \pi \in P\}$.
$\mathbb{P} \times \mathbb{Q}$	return Cartesian product $\{\alpha \cdot \beta \mid \alpha \in P \text{ and } \beta \in Q\}$.

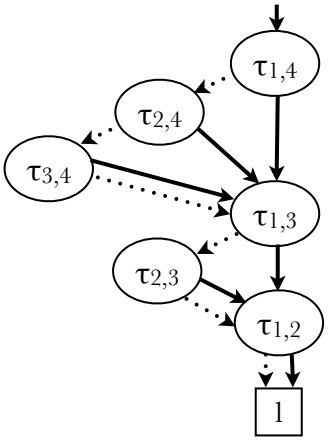


Figure 2.13. The πDD for S_4.

A permutation set S_n is a good example to demonstrate high compression of πDDs. Let $\mathbb{S}_n$ denote the πDD for S_n. We can recursively construct $\mathbb{S}_n$. Suppose we already obtained $\mathbb{S}_{n-1}$. We consider $(n-1)$-permutations as n-permutations with $\pi_n = n$. Thus, $\mathbb{S}_{n-1}.Swap(k,n)$ consists of all n-permutations π such that $\pi_n = k$. Therefore, $\mathbb{S}_n$ can be obtained by computing $\mathbb{S}_{n-1}.Swap(1,n) \cup \mathbb{S}_{n-1}.Swap(2,n) \cup \cdots \cup \mathbb{S}_{n-1}.Swap(n-1,n) \cup \mathbb{S}_{n-1}$. Algorithm 2.3.1 realizes this procedure by loops. Figure 2.13 shows $\mathbb{S}_4$. While the cardinality of S_n is $n!$, the number of internal nodes in $\mathbb{S}_n$ is $n(n-1)/2$ as shown in Figure 2.13.

Algorithm 2.3.1 Construct $\mathbb{S}_n$.

```
1:  procedure ConstructSn(n)
2:      S_0 ← the 1-sink
3:      for i = 1 to n do
4:          S_i ← S_{i−1}
5:          for j = 1 to i − 1 do
6:              S_i ← S_i ∪ S_{i−1}.Swap(j, i)
7:          end for
8:      end for
9:      return S_n
10: end procedure
```

Chapter 3

Applications of transposition-based πDDs

In this chapter, we give use cases of πDDs for permutation problems: reversible circuit debugging in Section 3.1 and cycle-type partition in Section 3.2. For each problem, we provide new algorithms utilizing πDDs and confirm effectiveness of πDDs by theoretical analyses and experiments.

3.1 Debugging Erroneous Reversible Circuit

Computation is *reversible* if we can determine an input pattern for a given output pattern. Reversible computation is a fundamental technology for next generation computation. The reversible property indicates that reversible computation is information-lossless. Therefore, reversible computation is used for *low power design* [9, 51] and *optical computing* [24]. In addition, *quantum computation* [63] is also related to reversible computation. Quantum computation exploits quantum mechanical phenomena such as superposition, entanglement, etc. and utilizes qubits rather than conventional bits for computation. Since quantum computation can be described as multiplication of unitary matrices, quantum computation is inherently reversible and thus design methods for reversible circuits are frequently utilized for automatic design of quantum circuits [6, 58, 73, 85].

Due to the reversible property, a *reversible circuit*, which is a circuit for reversible computation, has neither fan-out nor feedback, i.e. formed as a cascade of reversible logic gates. This distinguishes synthesis of reversible circuits from irreversible ones, and attracts many researchers to study synthesis approaches [20, 26, 55, 71, 83]. While synthesis of reversible circuits is a hot topic in the area of reversible computation, there are few results concerning debugging such circuits, which is another important process to analyze reversible circuits.

Wille et al. [84] proposed the first algorithm to debug reversible circuits using SAT formulation and SAT solvers based on debugging techniques for irreversible circuits. Frehse et al. [32] gave a simulation-based approach and combined it with the SAT-based approach. Tague et al. [77] provided another approach for a single gate error using πDDs. Since their methods consider only a single gate error, Jung et al. [43] proposed an extended approach for multiple gate errors. However, there are two problems to be considered:

- These algorithms use exponential algorithms or data structures, i.e. they are intractable in the worst case.
- These algorithms only detect error positions, i.e. cannot fix errors efficiently.

In this section, we address these tasks with different approaches for a single error and multiple errors, respectively.

3.1.1 Contribution Summary

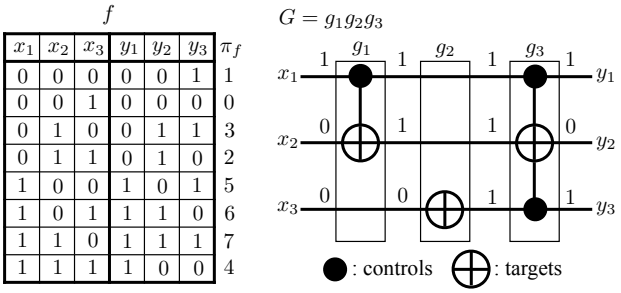
For a single error, we propose a theoretically improved debugging algorithm. This algorithm uses the lemma in [84] and new valid gate checking methods. For multiple errors, we provide a dynamic programming approach using πDDs. Although this algorithm has the worst-case complexity similar to the approach of Tague et al. [77] for a single error, it can fix multiple errors and debug them.

We evaluate the efficiency of our algorithms using computational experiments. For single error circuits, our algorithm achieves a significant improvement compared with previous approaches. For multiple error circuits, our algorithm succeeds to fix errors in circuits with few lines by the minimal corrections.

3.1.2 Reversible Functions, Circuits, and Gates

A function $f : \{0,1\}^n \to \{0,1\}^n$ is *reversible* if it is bijective, i.e., we can determine an input from the corresponding output. Hence, a function f is considered as a permutation on $\{0,1,\ldots,2^n-1\}$, by considering inputs as a binary number. We denote the permutation corresponding to a reversible function f by $\pi_f : \{0,1,\ldots,2^n-1\} \to \{0,1,\ldots,2^n-1\}$.

Reversible circuits realize reversible functions and consist of reversible gates. A reversible circuit for an n-bit Boolean function has n lines as shown on the right of Figure 3.1. Reversible circuits have no fan-out or feedback due to their reversible properties. Therefore, a reversible circuit is a cascade of reversible gates. Several reversible



x_1	x_2	x_3	y_1	y_2	y_3	π_f
0	0	0	0	0	1	1
0	0	1	0	0	0	0
0	1	0	0	1	1	3
0	1	1	0	1	0	2
1	0	0	1	0	1	5
1	0	1	1	1	0	6
1	1	0	1	1	1	7
1	1	1	1	0	0	4

Figure 3.1. Truth table of a reversible function f and a reversible circuit G realizing f.

gates have been invented to synthesize reversible circuits, such as Toffoli [79], Fredkin [31], and Peres [67] gates.

Let $L = [n]$ be a set of lines. *Toffoli gates* have multiple (possibly zero) *control lines* $C = \{c_1, \ldots, c_k\} \subset L$ and one *target line* $t \in L \setminus C$. For example, the Toffoli gate g_3 in Figure 3.1 has the control lines $C = \{1,3\}$ and the target line $t = 2$. A Toffoli gate inverts the target line when inputs for all the control lines are 1. Let x_i and y_i be the i-th line's input and output of a Toffoli gate, respectively. Then, we formally define Toffoli gates as follows:

$$\begin{aligned} y_t &= x_t \oplus x_{c_1} \cdots x_{c_k}, \\ y_i &= x_i \quad \text{if } i \neq t. \end{aligned}$$

Fredkin gates are similar to Toffoli gates, but have two target lines t_1, t_2 ($t_1, t_2 \notin C$) and swap target lines when all the control lines are 1. Fredkin gates are defined as follows:

$$\begin{aligned} y_{t_1} &= x_{t_1} \overline{x_{c_1} \ldots x_{c_{|C|}}} \vee x_{t_2} x_{c_1} \ldots x_{c_{|C|}} \\ y_{t_2} &= x_{t_2} \overline{x_{c_1} \ldots x_{c_{|C|}}} \vee x_{t_1} x_{c_1} \ldots x_{c_{|C|}} \\ y_i &= x_i \quad \text{if } i \neq t_1, t_2 \end{aligned}$$

Peres gates are also similar to the two gates, but little bit complicated:

$$\begin{aligned} y_{t_1} &= x_{t_1} \oplus x_{t_2} x_{c_1} \ldots x_{c_{|C|}} \\ y_{t_2} &= x_{t_2} \oplus x_{c_1} \ldots x_{c_{|C|}} \\ y_i &= x_i \quad \text{if } i \neq t_1, t_2 \end{aligned}$$

In this section, we mainly focus on debug of circuits that consists of Toffoli gates. Our method can be easily extend to Fredkin gates and Peres gates as discussed in Section 3.1.3 and 3.1.4.

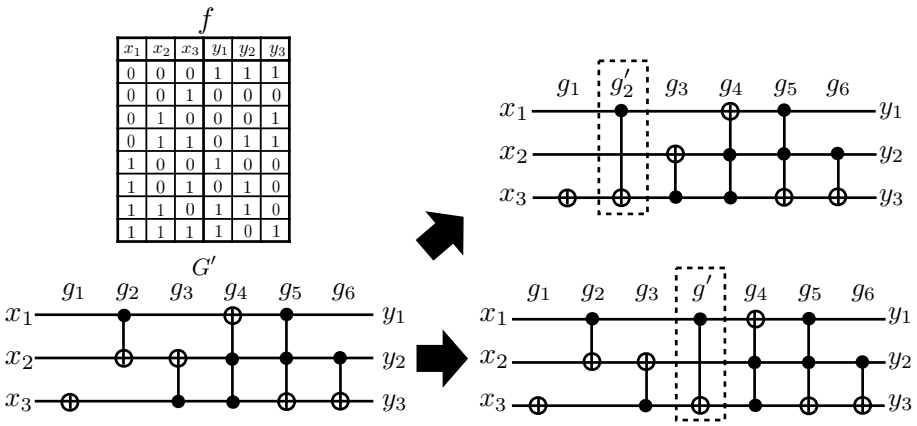


Figure 3.2. An erroneous circuit G' and two fixed circuits realizing f.

Since these gates themselves represents a reversible function, we can represent the function corresponding to a gate as a permutation. We denote by π_g the permutation corresponding to a gate g as well as a reversible function. Then the permutation representation π_G of the function realized by a reversible circuit $G = g_1 \cdots g_d$ equals $\pi_{g_1} \cdot \cdots \cdot \pi_{g_d}$.

3.1.3 Debugging Single Error

We define the single error debugging problem of reversible circuits. Let $f : \{0,1\}^n \to \{0,1\}^n$ be a reversible function and $G = g_1 \dots g_d$ be a reversible circuit with n lines such that $\pi_G = \pi_f$. We define G' to be a *single error circuit* for f if $\pi_{G'} \neq \pi_f$ and G' has:

- a replaced error: there is a gate $g' \neq g_i$ s.t. $G' = g_1 \dots g_{i-1} g' g_{i+1} \dots g_d$,
- an inserted error: there is a gate g' s.t. $G' = g_1 \dots g_{i-1} g' g_i g_{i+1} \dots g_d$, and
- a removed error: $G' = g_1 \cdots g_{i-1} g_{i+1} \cdots g_d$.

The goal of the single error debugging problems is to find the position of an error in an erroneous circuit G' and fix it in order to realize f correctly. We note that even if the number of embedded errors is only one, sometimes there are several ways to debug the circuit. For example, Figure 3.2 describes an erroneous circuit G' and an objective function f. At this instance, we have the two ways to debug G': replacing g_2 with g_2' or inserting g' between g_3 and g_4. In general, we cannot determine which of them the original error is. Therefore, we set our goal to list all the ways to debug G'.

Related Work

Wille et al. proposed a debugging method using SAT solvers [84]. They used *SAT* (*Boolean satisfiability*) formulation for debugging problems and solved it with SAT solvers. This method has three problems to be overcome:

- There are $O(nd)$ variables in SAT formula. Though state-of-the-art SAT solvers work practically fast, solving SAT is believed to require exponential time in the worst case. This is therefore not scalable for a large d.
- Their method can find only error candidates, which may include non-errors.
- Their method can debug only a replaced error.

We also note that this method requires verification preprocess to obtain some counterexamples.

Frehse et al. provided a simulation-based debugging algorithm [32]. Their method eliminates error candidates based on the fact that an error gate must be *activated* (i.e. all the inputs of control lines are 1) for all counterexamples. This method is fast because it runs in linear time with respect to the number of gates and lines. However, outputs of this method also can contain non-errors, since the activation property is a necessary condition but not a sufficient condition.

Tague et al. gave a debugging method using πDDs for a removed error [77]. They considered a gate as a permutation, and used πDDs to represent the set of gates. They insert a πDD into an erroneous circuit G' as an arbitrary gate, and calculate the compositions by Cartesian product operations. If the compositions contain f, it means G' has a removed error. This method also has two problems:

- The size of πDDs for a set of N-permutations is $O(2^{N^2})$, and now $N = 2^n$. It is not scalable for even small n.
- Their method can detect an error but cannot find its position and fix it.

We provide an algorithm overcoming these problems. More precisely, we propose a worst-case $O(n2^n d)$ time algorithm, which can find and fix all the three types of errors.

Proposed Method for Single Error

Our method is based on Lemma 3 in [84]:

Theorem 3.1.1 (Lemma 3 in [84])**.** *Let f be an objective reversible function and $G = g_1 \dots g_i \dots g_d$ be an erroneous circuit for f. Then G can be fixed by replacing any gate g_i of G with a cascade of gates specifying a function $\pi_{G_i^{fix}} = \pi_{g_{i-1}}^{-1} \dots \pi_{g_1}^{-1} f \pi_{g_d}^{-1} \dots \pi_{g_{i+1}}^{-1}$.*

This theorem states that, if $\pi_{G_i^{fix}}$ can be represented by a single Toffoli gate, the i-th gate is a replaced error and we can fix it by replacing it with the Toffoli gate corresponding to $\pi_{g_i^{fix}}$. Hereafter, we assume gates g_i are represented as permutations, and a cascade of gates means the composition of permutations. In other words, g_i is directly used instead of π_{g_i}. Then the single replaced error circuit problem can be solved as follows: checking whether $G_i^{rep} = g_{i-1}^{-1} \dots g_1^{-1} f g_d^{-1} \dots g_{i+1}^{-1}$ can be represented as a single Toffoli gate for all $1 \leq i \leq d$. Similarly, debugging problems for the other types of errors can be solved too:

- an inserted error : checking whether $G_i^{ins} = g_{i-1}^{-1} \cdots g_1^{-1} f g_d^{-1} \cdots g_{i+1}^{-1}$ can be represented as an identity permutation ι (with length 2^n) for all $1 \leq i \leq d$.
- a removed error : checking whether $G_i^{rem} = g_i^{-1} \cdots g_1^{-1} f g_d^{-1} \cdots g_{i+1}^{-1}$ can be represented as a single Toffoli gate for all $0 \leq i \leq d$.

Note that the position of a removed error is between two gates or two ends. We say a removed error occurs at the 0-th position if the error position is the left g_1, and at the i-th position if the error position is the right of g_i.

We let $N = 2^n$ for brevity. If we had an $O(h(n))$ time algorithm checking whether a given permutation represents a Toffoli gate, we could solve the single error circuit problem in $O(d(Nd + h(n)))$ by calculating the products G_i^{rep}, G_i^{ins}, and G_i^{rem} of $O(d)$ N-permutations and running a checking algorithm for all $0 \leq i \leq d$. We can improve this complexity by using the following properties:

- $G_i^{rep} = G_i^{ins}$,
- $G_i^{rem} = g_i^{-1} G_i^{ins}$,
- $G_i^{ins} = G_{i-1}^{rem} g_i$.

That is, incremental calculation of G_i^x from G_{i-1}^x costs only $O(N)$ time for N-permutation composition. Hence we can solve a single error circuit problem in $O(d(N + h(n)))$. Algorithm 3.1.1 gives the entire procedure.

The Toffoli gate checking problem is also solved in $O(nN)$ time by Algorithm 3.1.2. A permutation representing a Toffoli gate works as a transposition between integers a and b if a and b differ exactly a target bit, and all the control bits are 1. Lines 3–22 of Algorithm 3.1.2 identify control lines and a target line, eliminating cases that do not satisfy necessary conditions. Lines 24–31 check whether control lines and a target line work as an expected Toffoli gate.

This algorithm not only checks whether a given permutation can be specified by a single Toffoli gate, but also identifies the corresponding Toffoli gate. That is, we can

Algorithm 3.1.1 Debugging a single error circuit

```
1: procedure DEBUGSINGLEERROR(f, G)
2:     G_0^rem ← f g_d^{-1} g_{d-1}^{-1} ··· g_1^{-1}
3:     if ISTOFFOLI(G_0^rem) then
4:         Report a removed error: the gate G_0^rem was removed at the 0-th position.
5:     end if
6:     for i = 1 to d do
7:         G_i^ins ← G_{i-1}^rem g_i
8:         if G_i^ins = ι then
9:             Report an inserted error: g_i is an extra gate.
10:        else if ISTOFFOLI(G_i^ins) then
11:            Report a replaced error: g_i should be replaced with G_i^ins.
12:        end if
13:        G_i^rem ← g_i^{-1} G_i^ins
14:        if ISTOFFOLI(G_i^rem) then
15:            Report a removed error: the gate G_i^rem was removed at the i-th position.
16:        end if
17:    end for
18: end procedure
```

directly debug G' with the Toffoli gate returned by the procedure. It costs $O(nN)$ time and therefore we can solve the single error circuit problem in $O(nNd)$ time[1].

We can design checking algorithms for Fredkin gates and Peres gates similarly. Generally speaking, given a set of gates, we can solve the single error circuit problem in $O(d(N+h(n)))$ time if we have an $O(h(n))$ time checking algorithm for the gates. We also can easily adapt to deal with *negative control lines*. A Toffoli gate with positive and negative control lines inverts its output of the target line when the inputs of all the positive controls are 1 and all the negative controls are 0.

3.1.4 Debugging Multiple Errors

We extend the single error circuit problem to the multiple errors circuit problem. We define *k-error* circuits as circuits including k errors. Note that k errors can consist of different kinds of errors: replaced errors, inserted errors, and removed errors can be included together. We also note that k-error circuits may be debugged by less than k corrections. For example, two inserted errors of the same Toffoli gate at adjacent posi-

[1]If we assume w-bit word RAM model, we can improve it to $O(\lfloor \frac{n}{w} \rfloor Nd)$ by adopting bit parallel techniques to manage control lines C in Algorithm 3.1.2.

tions need not to be debugged, in other words these can be debugged by 0 corrections. In the multiple errors circuit problem, we set our goal to find the minimum corrections.

Algorithm 3.1.2 Checking whether a given permutation represents a Toffoli gate.

```
1:  procedure ISTOFFOLI(π)
2:      C ← {1,...,n}, T ← φ
3:      for i = 0 to N − 1 do
4:          if π_{π_i} ≠ i then                    ▷ π_i is neither i nor swapped with π_{π_i}
5:              return False
6:          end if
7:          if i and π_i are swapped then
8:              if i and π_i differ only the j-th bit in binary then
9:                  T ← T ∪ {j}
10:                 if |T| > 2 then      ▷ there are two or more candidates of target lines
11:                     return False
12:                 end if
13:             else                     ▷ there are two or more candidates of target lines
14:                 return False
15:             end if
16:             for j = 1 to n do
17:                 if the j-th bit of i in binary is 0 then
18:                     C ← C \ {j}                ▷ eliminate candidates of control lines
19:                 end if
20:             end for
21:         end if
22:     end for
23:       ▷ The Toffoli gate corresponding to π must have controls C and a target t ∈ T
24:     for i = 0 to N − 1 do
25:         if ∀j ∈ C, the j-th bit of π_i in binary is 1, but π_i = i then
26:             return False          ▷ all the controls are 1 but the target is not inverted
27:         end if
28:         if ∃j ∈ C, the j-th bit of π_i in binary is 0, but π_i ≠ i then
29:             return False          ▷ some controls are 0 but the target is inverted
30:         end if
31:     end for
32:     return True
33: end procedure
```

Related Work

Jung et al. [43] proposed a SAT based debugging algorithm for multiple errors, which is an extension of [32]. They used pruning based on the *hitting set problem* and encoded it into SAT formulation. Although their method can process large circuits, it has two problems to be considered:

- Their method can debug only replaced errors.
- Their method can detect only error candidates, which includes non-errors and cannot fix them directly.

In this section, we try to overcome these problems.

Naïve extension of Existing Method

Our proposed method for k-error circuits is derived from Tague's πDD-based approach for single error circuits [77]. For an inserted error, this approach tries to insert a πDD representing all available gates (e.g. all possible Toffoli gates) into all possible positions. It can be easily extend to replaced errors and removed errors. If we insert (or replace, remove) k πDDs for a set of available gates at all the k-subsets of positions, we can detect all error positions and error types. However, there are the following problems:

- The number of all combinations of k out of d positions are $\binom{d}{k} = O(d^k)$. Furthermore, we consider three types of errors for each position, i.e. there are 3^k ways of combinations of error types. That is, this algorithm requires $O(3^k d^{k+1})$ πDD operations.
- All error positions can be detected. However correct gates for replaced errors and removed errors cannot be determined.

We attack these problems with our algorithm proposed below.

Proposed Method for Multiple Errors

We propose a debugging algorithm requiring only $O(dk)$ πDD operations[2] for k-error circuits. Our approach uses dynamic programming calculating $S_{i,j}$, defined as a set of permutations representing functions which can be realized by the first j gates with i

[2]Note that each πDD operation costs exponential time in 2^{N^2} in the worst case.

errors. The minimum x such that $f \in S_{x,d}$ is the number of minimum corrections. We can calculate $S_{i,j}$ by the following recurrence relations:

$$S_{0,0} := \emptyset,$$
$$S_{i,j} := (S_{i,j-1} \times \{g_j\}) \cup (S_{i-1,j-1} \times L) \cup (S_{i-1,j} \times L) \cup S_{i-1,j-1},$$

where L is a set of available gates, which are Toffoli gates in our case. The first term represents non-error, the second term represents a replaced error, the third term represents an inserted error, and the last term represents a removed error is at the j-th position, respectively.

Since each $S_{i,j}$ is a set of permutations, we can use πDDs to represent them. Furthermore, calculation of recurrence relations requires only permutation set operations, union and Cartesian product, which are supported by πDDs. Each calculation of $S_{i,j}$ requires at most a constant number (i.e. 6) of operations. Hence this algorithm takes only $O(dk)$ πDD operations. In addition, we can calculate this recurrence relation by incrementing k. This means if the minimum corrections of a given k-error circuit is k', this algorithm only costs $O(dk')$ πDD operations, instead of $O(dk)$.

This algorithm can determine the minimum corrections, but cannot identify error positions and types yet. Error identification can be realized by starting from $S_{k',d}$ with a permutation f and reversely traversing to $S_{0,0}$. For example, if we now consider $S_{i,j}$ with a permutation π and $(\{\pi\} \times L^{-1}) \cap S_{i-1,j-1} \neq \phi$, where L^{-1} is the set of inverses of permutations in L, then a replaced error is detected at the position j. Furthermore, let $\pi' \in (\{\pi\} \times L^{-1}) \cap S_{i-1,j-1}$, we identify the original gate is $\pi \cdot \pi'^{-1}$. We then restart traversal from $S_{i-1,j-1}$ with π' until the first index is not 0.

3.1.5 Experiments

We implemented all algorithms in C++[3] and carried out experiments on a 3.20GHz CPU machine with 64GB memory. We randomly generate d Toffoli gates with n lines and concatenate them to make correct reversible circuits G. We prepare objective functions f for each circuit by simulating the circuit. Next, we generate erroneous reversible circuits G' with k errors based on correct circuits by repeating the following step k times: we randomly select a position and replace with a random gate, insert a random gate, or remove a gate.

Our implementation uses f and G' as inputs. For single error circuits, our implementation detects all corrections but only outputs the number of ways of corrections in

[3]Note that our implementation of Algorithm 3.1.2 uses bitwise operations of 64-bit integer (unsigned long long int in C++) to manage control lines C.

Table 3.1. Computation time (seconds) for single error circuits.

		d								
		10	50	100	500	1000	5000	10000	50000	100000
n	2	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.05	0.10
	4	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.09	0.17
	6	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.12	0.24
	8	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.21	0.41
	10	0.00	0.00	0.01	0.01	0.01	0.05	0.11	0.54	1.08
	12	0.00	0.00	0.01	0.03	0.04	0.21	0.40	2.05	3.99
	14	0.01	0.02	0.04	0.20	0.38	1.90	3.78	8.83	17.64
	16	0.03	0.10	0.19	0.89	1.75	8.78	17.61	87.71	149.00
	18	0.16	0.52	1.03	4.81	9.46	48.37	97.47	493.42	987.10
	20	0.60	1.87	3.88	18.28	35.90	187.28	377.66	—	—

order to reduce I/O time and concentrate the evaluation of the performance of our algorithm. For multiple error circuits, since the way of minimum corrections can be huge, our implementation detects only one way of minimum corrections and outputs it.

Experiments for Single Error

Computation time of Algorithm 3.1.1 for single error circuits (i.e. k=1) is shown in Table 3.1. This table shows that our algorithm is linear with the number of gates d and almost exponential with the number of lines n. It agrees with the theoretical complexity of our algorithm analyzed in Section 3.1.3.

In [84], the SAT solver-based algorithm takes about 2000 seconds or more for $n \geq 8$ and $d \geq 5000$ circuits. On the other hand, our algorithm takes under 1 second for circuits of such scale. Furthermore, in [77], the πDD-based algorithm takes more than 100 seconds for $n \geq 4$ and $d \geq 1000$ cases, while our algorithm takes under 0.01 seconds for these cases. This significant improvement is likely due to the theoretical improvement of our algorithm, and not simply to hardware and test case differences.

The simulation-based approach proposed by Frehse et al. in [32] seems to be faster than or equal to our algorithm: Their method completed simulation to detect error candidates in 20 seconds for the $n = 15$ and $d = 716934$ circuit. However, their method output over 30000 error candidates, including non-errors. This is impractical to check manually. In contrast, our algorithm returned only one correction for the $n = 16$ and $d = 100000$ erroneous circuit embedded a replaced error.

Experiments for Multiple Errors

We also carried out experiments for multiple error circuits. We randomly embedded k errors in circuits consisting of d gates with n lines. Figures 3.3–3.6 show experimental results for 1-, 2-, 3-, and 4-error circuits, respectively.

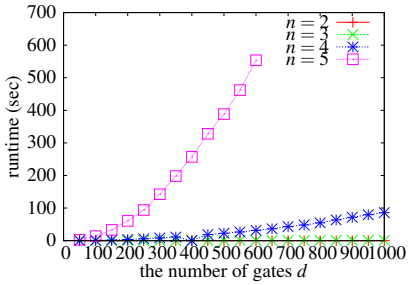


Figure 3.3. Runtime for debugging 1-error circuits.

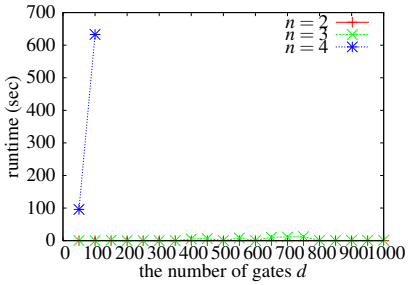


Figure 3.4. Runtime for debugging 2-error circuits.

For 1-error circuits in Figure 3.3, i.e. single error circuits, our πDD-based algorithm can perform in 1000 seconds for $n = 5$ and $d = 600$ circuits. For 2-error circuits in Figure 3.4, however, all the cases with $n = 5$ are time-outs even at $d = 50$. Almost all $n = 4$ cases also time-out; the algorithm can debug up to 100-gate circuits. Results in [43] show that the SAT based method is more scalable: e.g. this method can process $n = 8$ and $d = 637$ circuits in about 300 seconds. However, outputs of this method can include non-errors, and cannot fix them automatically. On the other hand, our method can fix them. For sufficiently small circuits, our method can provide richer debugging information.

Results of 3- and 4-error circuits in Figures 3.5 and 3.6. Our algorithm seems to be enough scalable for the circuits with $n \leq 3$. Debugging time for 3-errors and one of 4-errors seems similar. This is because in random circuits we prepared, the minimum correction of $n = 2$ circuits is usually 1, and for $n = 3$ circuits is usually 2, regardless of the number of embedded errors. In Figure 3.6, $d = 50$ and $d = 500$ in $n = 3$ cases seem to be somehow outliers. It is because the minimum correction size of the $d = 50$ circuit is 3, and for $d = 500$ circuit it is 1.

These results indicate that the minimum correction and the number of lines exponentially affect computation time. On the other hand, the number of gates seems to affect linearly for small gates ($n = 2, 3$), but affect quadratically or exponentially for slightly larger gates ($n = 4, 5$).

3.1.6 Concluding Remarks for Debugging Reversible Circuits

In order to debug erroneous reversible circuits, we propose two kinds of algorithms. The first one is an efficient method for circuits having at most one error. This method uses permutation properties of reversible gates and gate checking algorithms. This method can handle more general gate library if we achieve to design gate checking algorithms for the library. The efficient performance of this method is shown theoreti-

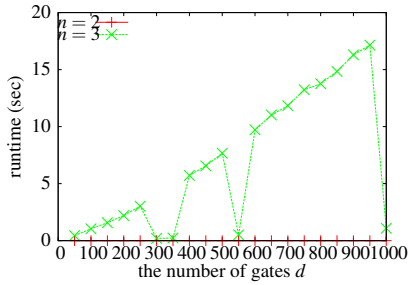


Figure 3.5. Runtime for debugging 3-error circuits.

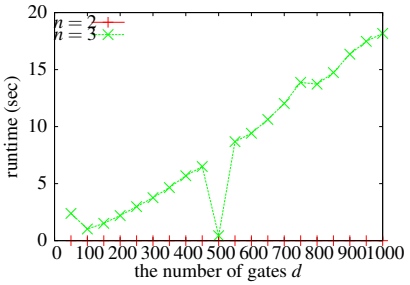


Figure 3.6. Runtime for debugging 4-error circuits.

cally and experimentally, comparing with existing methods. The second algorithm can debug multiple error circuits based on a dynamic programming approach and πDDs. Although the scalability of this algorithm is exponentially worse than the first one, the algorithm enables us to debug more general erroneous reversible circuits.

For future work, we would like to modify the first algorithm to handle circuits with garbage output lines. Garbage lines can output arbitrary values, i.e. multiple permutations can realize desired behavior. This means that multiple G_i's should be considered. Of course πDDs can handle this, but such an algorithm will lose the efficiency of our first approach.

For multiple errors, more scalable algorithms are desirable. We are also interested in the expected number of the minimum corrections for circuits with n lines, d gates, and k randomly-embedded errors. From experimental results, we guess that minimum correction tend to become relatively small with the number of embedded errors. If we show that the size is sufficiently small with high probability, perhaps we need not to consider debugging circuits with a large number of errors.

3.2 Cycle-type Partition of Permutation Sets

Cycle-type is an index for permutations. The cycle-type of a permutation π is a vector $\vec{c^{\pi}}$ such that the l-th element c_l^{π} indicates the number of l-cycles in π. We can partition a permutation set into equivalence classes of permutations such that each class consists only of permutations having the same cycle-type. This is *cycle-type partition* of a permutation set.

Cycle-types are useful for analysis of permutation sets. For example, the *cycle index* of a permutation set P, which is a polynomial $\sum_{\pi \in P} \prod_{k=1}^{n} x_k^{c_k^{\pi}}$, is used as a generating function for the Pólya enumeration theorem [86]. Another instance of applications is complexity analysis of algorithms processing permutations, e.g., sorting. Thus, calculation of the cycle-type class partition for a given permutation set contributes basic understandings of mathematical characterization of permutation sets.

In this section, our goal is computation of the cycle-type class partition for a given permutation set. Since calculation of the cycle-type of a permutation runs in time linear in the length of the permutation, we can classify each permutation according to its cycle-type one by one, if a given permutation set can be stored on memory in naïve array representation. On the other hand, we sometimes know only some characteristics of a permutation set, e.g. generators or restrictions for a permutation set, rather than an explicit list of permutations because the cardinality of the permutation set is too large to store all of them on memory. In such cases, the naïve algorithm by one-by-one classification will not work in feasible time.

3.2.1 Contribution Summary

We propose an algorithm to overcome this issue by compression with πDDs. This algorithm supposes that an input permutation set is given in the form of a πDD, and directly constructs the πDDs corresponding to the partitioned sets. Hence if a given permutation set can be compactly represented by a πDD, this algorithm can work well even if the cardinality of the set is huge. This feature may also enable us to calculate a cycle-type partition faster and less memory, and manipulate and analyze each set in the partition by using πDD operations on memory.

In [87], Yamada and Minato have proposed another method to construct πDDs representing a cycle-type partition. The existing method must construct the πDDs for the cycle-type partition of S_n regardless of a given set, where n is the length of permutations in a given permutation set. On the other hand, our method does not require other than the πDD for the input set. This may discriminate between our algorithm and the existing method in terms of time and space efficiency in the cases such that computation of the cycle-type partition for S_n is the bottleneck of the existing method. In addition, our

method can construct the πDDs for the cycle-type partition of S_n faster than the existing method, as shown by the experimental results in Section 3.2.4. This means our method can be used to accelerate the existing method by replacing the construction algorithm in the existing method with our method.

3.2.2 Definitions and Notations of Cycle-types

We have introduced the cycle decomposition of permutations in definition 2.1.7.

Definition 3.2.1. *The* cycle-type *of an n-permutation π is a vector $\vec{c^{\pi}}$ with length n such that the i-th element c_i^{π} of $\vec{c^{\pi}}$ is the number of i-cycles in the cycle decomposition of π.*

For example, a permutation $\pi = (4,6,1,3,7,2,5)$ is decomposed into $(2\,6)(5\,7)(1\,4\,3)$, thus $\vec{c^{\pi}} = (0,2,1,0,0,0)$ because the cycle decomposition of π has two 2-cycles and one 3-cycle. The cycle-type of permutations defines an equivalence relation on permutations.

Definition 3.2.2. *Two permutations π and σ are* cycle-type equivalent*, noted as $\pi \sim_{cy} \sigma$, if their cycle-types are the same, i.e. $\vec{c^{\pi}} = \vec{c^{\sigma}}$.*

For example, two permutations $\pi = (4,6,1,3,5,2)$ and $\sigma = (1,4,6,2,3,5)$, which are decomposed into $(5)(2\ 6)(1\ 4\ 3)$ and $(1)(2\ 4)(3\ 6\ 5)$ respectively, are cycle-type equivalent because they have the same cycle-type $(1,1,1,0,0,0)$.

Definition 3.2.3. *The* cycle-type partition *of a permutation set P is the equivalence partition of P with respect to the cycle-type equivalence $\sim_{cy}$.*

Note that the number of cycle-types of n-permutations is equals to the number of integer partitions of n, because the sum of the elements in $\vec{c^{\pi}}$ is always n. This means the number of sets in cycle-type partition for an n-permutation set is at most the number of integer partitions of n.

We also provide the definition of *conjugacy classes* to introduce the existing method by Yamada and Minato [87].

Definition 3.2.4. *Two permutations π and σ in a permutation group G are* conjugate *if there exists $g \in G$ such that $\pi = g \cdot \sigma \cdot g^{-1}$. This can be considered as binary relation $\pi \sim_{cj} \sigma$. The* conjugacy classes *of G are defined as equivalence classes of G with respect to $\sim_{cj}$.*

The conjugacy classes of S_n has an important characteristic related to cycle-type.

Theorem 3.2.1. *Each conjugacy class of S_n has one-to-one correspondence with a class in the cycle-type partition of S_n. In other words, the conjugacy classes of S_n is equals to the cycle-type partition of S_n.*

Proof. This is followed by Lemma 1 in Appendix of [87]. □

Note that this relation does not hold for a general permutation set.

3.2.3 Cycle-type Partition Algorithm

Before proceeding to our algorithm, we briefly review two algorithms compared with our algorithm in our experiments. Next, we provide our proposed algorithm.

Previous Methods

The first algorithm is a naïve method: for each permutation in a given permutation set, compute its cycle-type in linear time and assign it into the corresponding class. The complexity of the naïve method is $O(n|P|)$, where P is a given permutation set and n is the maximum length of permutations in P.

The second algorithm is proposed by Yamada and Minato in [87]. This algorithm uses πDDs as input and output like our proposed method. Yamada's algorithm consists of two steps: At first, it constructs πDDs corresponding to the cycle-type partition of S_n. Next it computes intersections of a given permutation set P and each cycle-type class of S_n with πDD operations. From Theorem 3.2.1, the cycle-type partition of S_n is obtained by computation of the conjugacy classes of S_n. Hence, they proposed an algorithm to compute the conjugacy classes of a given permutation group G, and apply it to S_n for this purpose. They also propose sophisticated techniques based on group theory to reduce the runtime of the algorithm. Unfortunately, it is difficult to analyze the complexity of their method since the number of loops in their algorithm seems not to be obvious.

Proposed Method

Our proposed method constructs objective πDDs in bottom-up manner, traversing the input πDD. To achieve this, the proposed method uses a relation between transpositions and cycle decompositions.

Proposition 3.2.1. *Let* $\pi = (a_{1,1}\ \ldots\ a_{1,l_1})\cdots(a_{k,1}\ \ldots\ a_{k,l_k})$ *be a permutation. A transposition with two elements in different cycles merges the two cycles into one cycle. More precisely,* $\pi \cdot \tau_{a_{x,i},a_{y,j}} = \cdots(a_{x,1}\ \ldots\ a_{x,i-1}\ a_{y,j}\ \ldots\ a_{y,l_y}\ a_{y,1}\ \ldots\ a_{y,j-1}\ a_{x,i}\ \ldots\ a_{x,l_x})\cdots$. *On the other hand, a transposition with two elements in the same cycle divides the cycle into two cycles. More precisely,* $\pi \cdot \tau_{a_{x,i},a_{x,j}} = \cdots(a_{x,1}\ \ldots\ a_{x,i-1}\ a_{x,j}\ \ldots\ a_{x,l_x})\,(a_{x,i}\ \ldots\ a_{x,j-1})\cdots$.

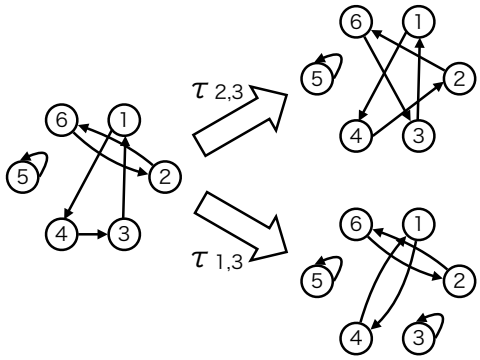


Figure 3.7. Union and separation of cycles by a transposition

An example for this proposition is shown in Figure 3.7.

Here, we consider about πDD's transpositions. Let $\mathbb{P}_1$ be a sub-πDD pointed by the 1-edge from a node labeled with $\tau_{i,j}$. Then, this represents the compositions of all the permutations in the set represented by $\mathbb{P}_1$ and $\tau_{i,j}$. Here, all the permutation π in $\mathbb{P}_1$ satisfy $\pi_j = j$, since the transposition decomposition of π must not include $\tau_{x,y}(x, y \geq j)$ due to the transposition ordering rule of πDDs. This means all the permutations in $\mathbb{P}_1$ have the cycle consisting only of j. Therefore, $\tau_{i,j}$ merges the cycle including i and the cycle (j) in all the permutations in $\mathbb{P}_1$ according to Proposition 3.2.1. Namely, a transposition sequence on a πDD path represents a series of processes adding an element to one of the cycles.

We introduce a new equivalence relation to utilize this property.

Definition 3.2.5. *The cycle-set of a permutation* $\pi = (a_{1,1} \dots a_{1,l_1}) \cdots (a_{k,1} \dots a_{k,l_k})$ *is the set* $\{\{a_{1,1} \dots a_{1,l_1}\}, \dots, \{a_{k,1} \dots a_{k,l_k}\}\}$*, i.e., the family of sets each of which consists of all the elements of a cycle in* π*. Two permutations* π *and* σ *are* cycle-set equivalent *if they has the same cycle-set. The* cycle-set partition *of a permutation set is an equivalence partition with respect to cycle-set equivalence, and a cycle-set class is a set in a cycle-set partition.*

Then, applying $\tau_{i,j}$ to cycle-set equivalent permutations in $\mathbb{P}_1$ yields cycle-set equivalent permutations due to the property. Hence, we can design a recursive algorithm to construct πDDs corresponding to the cycle-set partition, as shown in Algorithm 3.2.1. In Algorithm 3.2.1, $\mathcal{P}$ denotes the family of πDDs and $\mathcal{P}_{\mathfrak{S}}$ means a πDD representing a cycle-set class for a cycle-set $\mathfrak{S}$ in $\mathcal{P}$. Algorithm 3.2.1 is recursively called for 0-child and 1-child, and obtain the cycle-set partitions of them respectively. Then, it unites the two cycle-set partitions, taking the union of πDDs with the same cycle-set in the two cycle-set partitions.

The cycle-set partition of a given permutation set is useful to computing cycle-type partition due to the following proposition.

Algorithm 3.2.1 Algorithm calculating the family of πDDs representing the cycle-set partition of the permutation set represented by πDD $\mathbb{P}$

procedure CYCLESETPARTITION($\mathbb{P}$)
  Let $\mathcal{P} = (\tau_{i,j}, \mathbb{L}, \mathbb{R})$
  $\mathcal{L} \leftarrow$ CYCLESETPARTITION($\mathbb{L}$)
  $\mathcal{R} \leftarrow$ CYCLESETPARTITION($\mathbb{R}$)
  for all $\mathfrak{S} \in$ set partitions of $\{1, \ldots, j-1\}$ **do**
    $\mathfrak{A} \leftarrow \mathfrak{S} \cup \{\{j\}\}$
    $\mathcal{P}_{\mathfrak{A}} \leftarrow \mathcal{P}_{\mathfrak{A}} \cup \mathcal{L}_{\mathfrak{S}}$
    $\mathfrak{B} \leftarrow$ the family of sets s.t. j is added to the set including i in $\mathfrak{S}$
    $\mathcal{P}_{\mathfrak{B}} \leftarrow \mathcal{P}_{\mathfrak{B}} \cup (\tau_{i,j}, \text{the 0-sink}, \mathcal{R}_{\mathfrak{S}})$
  end for
  return $\mathcal{P}$
end procedure

Algorithm 3.2.2 Algorithm calculating the family of πDDs representing the cycle-type partition of the permutation set represented by πDD $\mathbb{P}$

procedure CYCLETYPEPARTITION($\mathbb{P}$)
  $\mathcal{P} \leftarrow$ CYCLESETPARTITION($\mathbb{P}$)
  for all $\mathfrak{S} \in$ set partitions of $\{1, \ldots, j-1\}$ **do**
    $\vec{c} \leftarrow$ a vector s.t. c_k is the number of sets with the cardinality k in $\mathfrak{S}$
    $\mathcal{X}_{\vec{c}} \leftarrow \mathcal{X}_{\vec{c}} \cup \mathcal{P}_{\mathfrak{S}}$
  end for
  return $\mathcal{X}$
end procedure

Proposition 3.2.2. *All the permutations in a cycle-set class are cycle-type equivalent.*

Thus, it is sufficient to merge cycle-set classes obtained by Algorithm 3.2.1 with the same cycle-type into a cycle-type class. Algorithm 3.2.2 provides the procedure to achieve this.

The complexity of Algorithm 3.2.1 is bounded by the sum of the cardinality of the family in each step. The cardinality of the family is bounded by the number of set partitions of $\{1, \ldots, j\}$ for $\tau_{i,j}$, and it is known as Bell number B_j. Therefore, the complexity of Algorithm 3.2.1 is roughly bounded by $O(|\mathbb{P}|B_n)$ for a πDD $\mathbb{P}$ representing a set of n-permutations, where $|\mathbb{P}|$ is the size of $\mathbb{P}$. However, in practice, a given permutation set may not include permutations with some of cycle-sets, and then the algorithm will work faster than the theoretical estimation. On the other hand, Algorithm 3.2.2 compute the union of πDDs at most B_n times. Unfortunately, because it is hard to estimate the

Table 3.2. Results of cycle-type partition algorithms for S_n

	Naïve		Yamada and Minato [87]		Proposed	
n	time (sec)	memory (MB)	time (sec)	memory (MB)	time (sec)	memory (MB)
8	0.01	4.6	0.31	35.8	0.02	34.4
9	0.11	30.0	2.70	269.8	0.12	56.0
10	1.09	297.5	21.50	1182.3	1.25	280.7
11	12.52	3821.6	169.88	9047.8	10.21	2053.9
12	154.14	44972.8	1479.63	64558.5	78.56	15035.3

complexity of union operations of πDDs, the entire complexity of Algorithm 3.2.2 is not obvious.

3.2.4 Experimental Results

We conducted computational experiments to evaluate the practical performance of our algorithm, compared with the other methods. We implemented all the algorithms in C++ with gcc 4.9.3 compiler. We carried out experiments on a 3.20 GHz CPU machine with 64 GB memory.

Table 3.2 shows the performance of algorithms for S_n. Our method is 20-fold faster and 5-fold less memory usage than Yamada's method. Comparing with naïve method, while our method is less efficient for $n \leq 10$ cases, our method is 2-fold faster and 3-fold less memory usage. The bottleneck of all the algorithms is memory usage: all the algorithm cannot work in $n = 13$ case due to memory shortage.

3.2.5 Concluding Remarks for Cycle-Type Partition

In this section, we use πDDs to partition a given permutation set with respect to cycle-type equivalence relation. Computational experiments indicate our method is more efficient than previous methods especially for lager n.

Since large memory usage is the bottleneck of algorithms including our method, improvement of space efficiency is most important future work. We also aim to apply our techniques in this paper to the conjugacy class partition. In addition, we will try to use constructed πDDs for practical and/or mathematical applications related to cycle-types, such as counting and analyses.

Chapter 4

Rotation-based πDDs and Their Applications

4.1 Rotation-based πDDs

To derive πDDs from ZDDs, we use transposition decomposition of permutations and assign transpositions to ZDD nodes as labels. Because there are many other ways to decompose permutations, we can use another decomposition to design permutation decision diagrams. Here, we focus on *left-rotations*.

Definition 4.1.1. *A* left-rotation $\rho_{i,j}$ $(i < j)$ *is a permutation* $(1,\ldots,i+1,i+2,\ldots,j-1,j,i,j+1,\ldots,n)$. *That is,*

$$\rho_{i,j}(k) = \begin{cases} i & \text{if } k = j \\ k+1 & \text{if } i \leq k < j \\ k & \text{otherwise.} \end{cases}$$

Example 4.1.1. *Composition of $\rho_{l,r}$ and an n-permutation π achieves the left-rotation in the interval $[l,r]$ of π, i.e., $\rho_{l,r} \cdot \pi = \pi_1 \ldots \pi_{l+1} \ldots \pi_r \pi_l \ldots \pi_n$.*

Left-rotations also uniquely decompose a permutation as follows.

Definition 4.1.2. *Left-rotation decomposition of a permutation π is defined as follows: We start with $\pi' = \iota$ and repeatedly apply left-rotations to π' in order to obtain π. At the k-th step, we apply left-rotation $\rho_{i,n+1-k}$, where i is the position of π_{n+1-k} in π', i.e., $\pi'_i = \pi_{n+1-k}$, and update $\pi' \leftarrow \rho_{i,j} \cdot \pi'$. The sequence of left-rotations to obtain π from ι is the left-rotation decomposition of π.*

Example 4.1.2. *For example, consider to decompose $(4,3,1,5,2)$ into a sequence of left-rotations. We start with $\pi' = \iota = (1,2,3,4,5)$. At the 1st step, we want to move 2 from the 2nd position to the 5th position. Thus, we obtain $\pi' = \rho_{2,5} \cdot \iota = (1,3,4,5,2)$.*

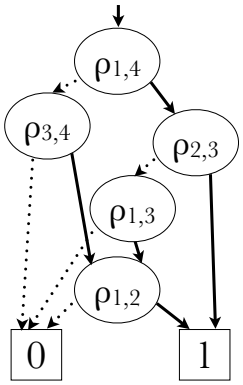


Figure 4.1. The Rot-πDD for $\{(2,1,4,3),(2,4,3,1),(4,3,2,1)\} = \{\rho_{1,2}\cdot\rho_{3,4}, \rho_{2,3}\cdot\rho_{1,4}, \rho_{1,2}\cdot\rho_{1,3}\cdot\rho_{1,4}\}$.

At the 2*nd step, we move* 5 *from the* 4*th position to the* 4*th position, i.e., we have not to rotate. At the* 3*rd step, we want to move* 1 *from the* 1*st position to the* 3*rd position. Thus, we obtain* $\pi' = \rho_{1,3}\cdot\rho_{2,5} = (3,4,1,5,2)$. *At the* 4*th step, we want to move* 3 *from the* 1*st position to the* 2*nd position. Thus, we obtain* $\pi' = \rho_{1,2}\cdot\rho_{1,3}\cdot\rho_{2,5} = (4,3,1,5,2) = \pi$. *Hence we obtain the left-rotation decomposition* $\rho_{1,2}\cdot\rho_{1,3}\cdot\rho_{2,5}$ *of* $\pi = (4,3,1,5,2)$.

Here, we propose *Rotation-based πDD* (*Rot-πDD*) by using left-rotation decomposition instead of transposition decomposition. Ordering of left-rotations is in the same manner for transpositions: $\rho_{x_1,y_1} < \rho_{x_2,y_2}$ if $y_1 > y_2$ holds, or $y_1 = y_2$ and $x_1 < x_2$ holds. Figure 4.1 is an example of a Rot-πDD for a permutation set $\{(2,1,4,3),(2,4,3,1),(4,3,2,1)\}$.

Rot-πDDs also have operations same as πDDs, e.g. union, intersection, and set difference. On the other hand, we should redesign an operation for $P\times\{\rho_{l,r}\}$ instead of $\mathbb{P}$.Swap(x,y) of πDDs for $P\times\{\tau_{x,y}\}$, because this operation is used in Cartesian product operation, which is useful in many applications. (For details of algorithms for the Swap and the Cartesian product operation of πDDs, please see [60].) We refer the Rot-πDD operation for $P\times\{\rho_{l,r}\}$ as $\mathbb{P}$.LeftRot(l,r).

As a first step, we assume that a permutation set P is a singleton $\{\pi\}$ and π is decomposed into left-rotations $\rho_{x_1,y_1}\ldots\rho_{x_k,y_k}$. We should compute left-rotation decomposition of the new permutation $(\rho_{x_1,y_1}\ldots\rho_{x_k,y_k})\cdot\rho_{l,r} = \rho_{x_1,y_1}\ldots\rho_{x_k,y_k}\rho_{l,r}$. If $y_k < r$, it is already left-rotation decomposition and thus there is nothing to do anymore. Otherwise, we should reform the composition of left-rotations. If we can transform $\rho_{x_k,y_k}\rho_{l,r}$ into $\rho_{l',r'}\rho_{x'_k,y_k}$ with $r' < y_k$, we achieve to obtain left-rotation decomposition by repeating to transform $\rho_{x_i,y_i}\rho_{l',r'}$ for $i < k$ until $r' \le y_i$ holds. In fact, such a transformation rule exists:

Theorem 4.1.1. *Let* $1 \le x < y \le n$, $1 \le l < r \le n$, *and* $r \le y$. $\rho_{x,y}\rho_{l,r}$ *can be transformed into the form of* $\rho_{l',r'}\rho_{x',y}$ *with* $r' < y$.

Proof. We consider four cases distinguished by the relation between l, x, and y.

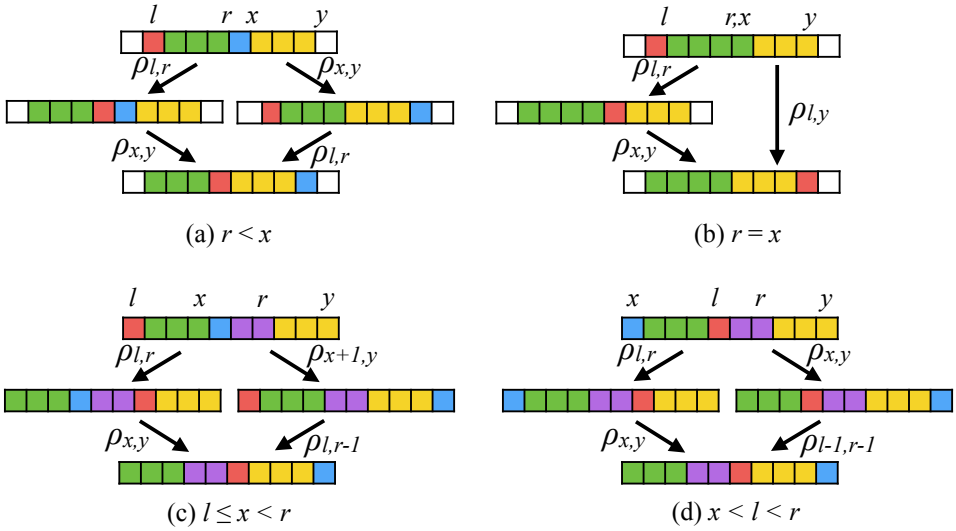


Figure 4.2. Transformation of compositions of two left-rotations

- If $r < x$, the intervals $[l,r]$ and $[x,y]$ are distinct, thus simply $\rho_{x,y}\rho_{l,r} = \rho_{l,r}\rho_{x,y}$.
- If $r = x$, $\rho_{l,r}$ moves the l-th element to the $r = x$-th position, and then $\rho_{x,y}$ moves it to the y-th position. All the other elements in $[l,y]$ are shifted to left by one. Thus $\rho_{x,y}\rho_{l,r} = \rho_{l,y}$[1].
- If $l \leq x < r$, $\rho_{l,r}$ moves the $x+1$-th element to the x-th position and $\rho_{x,y}$ moves it to the y-th position. $\rho_{l,r}$ moves the l-th element to the r-th position and $\rho_{x,y}$ moves it to left by one. The elements in $[l+1,x]$ and $[r+1,y]$ are shifted to left once by $\rho_{l,r}$ and $\rho_{x,y}$, respectively, and the elements in $[x+2,r]$ are shifted to left twice. This moves is simulated by $\rho_{l,r-1}\rho_{x+1,y}$.
- If $x < l < r$, $\rho_{l,r}$ fix the x-th element and $\rho_{x,y}$ moves it to the y-th position. $\rho_{l,r}$ moves the l-th element to the r-th position, and $\rho_{x,y}$ moves it to left by one. The elements in $[x+1,l-1]$ and $[r+1,y]$ are shifted to left once by $\rho_{l,r}$ and $\rho_{x,y}$, respectively, and the elements in $[l+1,r]$ are shifted to left twice. This moves is simulated by $\rho_{l-1,r-1}\rho_{x,y}$.

All the cases are visualized in Figure 4.2 for intuitive understandings. □

Based on Theorem 4.1.1, we can design a recursive algorithm to calculate the Rot-πDD for $P \times \{\rho_{l,r}\}$ from a Rot-πDD $\mathbb{P}$ for P. Let $\mathbb{P} = (\rho_{x,y}, \mathbb{P}_0, \mathbb{P}_1)$. This means that a set of the permutation whose left-rotation decomposition ends with $\rho_{l,r}$ in $\mathbb{P}$ is exactly represented by $\mathbb{P}_1 \times \{\rho_{x,y}\}$, and the other is corresponding to $\mathbb{P}_0$. Thus, $\mathbb{P}$.LeftRot(l,r) is obtained by $(\rho_{x',y}, \mathbb{P}_0$.LeftRot$(l,r), \mathbb{P}_1$.LeftRot$(l',r'))$. Algorithm 4.1.1 describes the entire procedure.

[1] We can consider $\rho_{l,y}$ is in the form of $\rho_{l',r'}\rho_{x',y}$ by introducing a dummy left-rotation $\rho_{1,1}\rho_{l,y}$.

Algorithm 4.1.1 Compute a Rot-πDD for $P \times \{\rho_{l,r}\}$ $(l < r)$.

procedure LEFTROT($\mathbb{P}, l, r$)
  if $\mathbb{P}$ is the 0-sink **then**
    return the 0-sink
  else if $\mathbb{P}$ is the 1-sink **then**
    return ($\rho_{l,r}$, the 0-sink, the 1-sink)
  end if
  if the result of LEFTROT($\mathbb{P}, l, r$) is memorized on cache **then**
    return the memorized result
  end if
  Let $\mathbb{P} = (\rho_{x,y}, \mathbb{P}_0, \mathbb{P}_1)$.
  if $y < r$ **then**
    return ($\rho_{l,r}$, the 0-sink, $\mathbb{P}$)
  end if
  $\mathbb{P}'_0 \leftarrow$ LEFTROT($\mathbb{P}_0, l, r$)
  if $r < x$ **then**
    $x' \leftarrow x$, $\mathbb{P}'_1 \leftarrow$ LEFTROT($\mathbb{P}_1, l, r$)
  else if $r = x$ **then**
    $x' \leftarrow l$, $\mathbb{P}'_1 \leftarrow \mathbb{P}_1$
  else if $l \leq x$ **then**
    $x' \leftarrow x+1$, $\mathbb{P}'_1 \leftarrow$ LEFTROT($\mathbb{P}_1, l, r-1$)
  else
    $x' \leftarrow x$, $\mathbb{P}'_1 \leftarrow$ LEFTROT($\mathbb{P}_1, l-1, r-1$)
  end if
  Memorize ($\rho_{x',y}$, $\mathbb{P}'_0$, $\mathbb{P}'_1$) on cache
  return ($\rho_{x',y}$, $\mathbb{P}'_0$, $\mathbb{P}'_1$)
end procedure

In the following sections, we will see the impact of Rot-πDDs with three instances: enumeration of Eulerian trails, enumeration of topological orders, enumeration of pattern-avoiding permutations. Utilizing Rot-πDDs makes algorithms empirically and theoretically more efficient than πDDs on the applications.

4.2 Enumeration of Eulerian Trails

Eulerian trails are trails passing through every edge in a graph exactly once. *Eulerian circuits* are defined as Eulerian trails that start and end at the same vertex. Eulerian paths have been first introduced by Euler [30] to solve "Seven Bridges of Königsberg problem" in 1736. Eulerian trails are attractive not only in mathematical points of view, also have industrial applications such as DNA fragment assembly [68] and CMOS circuit design [72].

Hierholzer and Wiener [36] have proposed an algorithm finding an Eulerian trail in time linear in given graph size. For counting the number of Eulerian trails in a given directed graph, there is a polynomial time algorithm called *BEST algorithm* based on matrix tree theorem [80]. On the other hand, for undirected graphs, the counting problem is known as a #P-complete problem [15]. Approximation algorithms for several undirected graph classes have been proposed [57]. For exactly counting, we can use a naïve backtracking search algorithm and improve its time complexity by using dynamic programming technique, which is still an exponential time algorithm. The enumeration problem for Eulerian trails is also considered. Kikuchi has proposed a linear time delay algorithm for enumeration of Eulerian trails [90], which output an Eulerian trails in $O(m)$ time after the previous output of another Eulerian trail.

In this section, we tackle enumeration of Eulerian trails in a given undirected graph. Since the number of Eulerian trails are exponential, we aim to store the Eulerian trails in the compressed data structure. We can use Rot-πDDs for this purpose because an Eulerian trail is a permutation of edges in a graph: each edge appears in a trail exactly once.

4.2.1 Contribution Summary

We propose an algorithm directly constructing a Rot-πDD representing the set of Eulerian trails, namely permutations of edges, in a given graph. We intentionally use Rot-πDDs instead of πDDs because the time and space complexity of the Rot-πDD-based algorithm can be well-bounded theoretically; as shown in Section 4.2.3, the time and space complexity of our algorithm is $O(m2^m)$, which is significantly less than the maximum number $m!$ of the Eulerian trails. We also conduct computational experiments to evaluate the performance of our algorithm. Experiments show that our algorithm succeeded to construct a Rot-πDD representing 3×10^{13} Eulerian trails of a random graph in 3 minute with 5×10^7 nodes. This means our method achieved six orders of magnitude compression.

4.2.2 Eulerian Trails

An *Eulerian trail* in a given graph is a sequence $(e_1, \ldots, e_m)$ of edges that composes a trail and includes all edges exactly once, i.e., $e_i \neq e_j$ if $i \neq j$. This implies an Eulerian trail corresponds to a (restricted) permutation of edges in a given graph. Note that an Eulerian trail is a trail, thus it may pass through a vertex multiple times. Especially, if the start vertex and the end vertex is the same one in an Eulerian trail, it is called as an *Eulerian circuit*.

We assume that a given graph is undirected, and has no self-loop and no isolated vertex.[2] In general, an undirected graph may have neither Eulerian trail nor Eulerian circuit. An undirected graph is *Eulerian* if it has at least one Eulerian circuit, while an undirected graph is *semi-Eulerian* if it has at least one Eulerian trail. It is the well-known property that a graph is Eulerian if and only if the graph is connected and every vertex has even degree. The property of semi-Eulerian graphs is also well-known: a graph is semi-Eulerian if and only if the graph is connected and exaclty two vertices only have odd degree. This means we can examine whether a given graph has at least one Eulerian trail or not in linear time. We hence assume that a given graph is Eulerian or semi-Eulerian.

If a given graph is Eulerian, there are multiple equivalent Eulerian circuits: for an Eulerian circuit $(e_1, \ldots, e_m)$, a circuit $(e_{i+1}, \ldots, e_m, e_1, \ldots, e_i)$ is also an Eulerian circuit. Therefore, we enumerate Eulerian circuits that start with a certain fixed edge, and we call it e_1. In addition, we use e_1 in a fixed direction, and we call the start vertex v_1. If a given graph is semi-Eulerian, the reverse of an Eulerian trail is also an Eulerian trail. Therefore, we enumerate Eulerian trails that start with a certain fixed vertex, which is one of two vertices with odd degree, and we call it v_1.

We denote the set of Eulerian trails (for semi-Eulerian case) and circuits (for Eulerian case) in a graph G by $eu(G)$, and the number of Eulerian trails and circuits by $EU(G) = |eu(G)|$. For example, the graph G in Figure 4.3 has $EU(G) = 6$ Eulerian trails with a start vertex b: $eu(G) = \{(1,2,3,4,6,5), (1,2,5,6,4,3), (3,2,1,4,6,5), (3,5,6,4,1,2), (4,6,5,2,3,1), (4,6,5,3,1,2)\}$.

A simple algorithm to enumerate all Eulerian trails is backtracking. At the beginning, we fix a start vertex $v = v_1$ and set v as a current vertex. Next we choose an edge $e = \{v,u\}$ from edges incident to v, add e to the used-edge list, and set u as a current vertex. We recursively traverse the graph without using used-edges again. When the recursive traversal is done, we restore the search state, i.e., set v as a current vertex and remove e from the used-edge list, and try other edges incident to v one by one. This backtracking algorithm finds an Eulerian trail in $O(m)$ time[3]. Since $EU(G)$ is $m!$ in the

[2] We can easily extend our proposed algorithm to be able to process directed graphs and/or graphs with self-loops and isolated vertices.

[3] More precisely, the complexity is bounded by $O(n+m)$. Here, we now assume that a given graph is

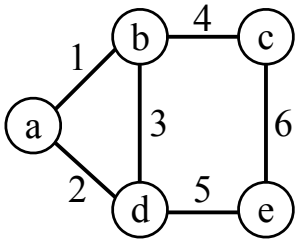


Figure 4.3. An example of semi-Eulerian graphs

worst case, this algorithm runs in $O(m \cdot m!)$.

We can improve the time complexity of this algorithm if we only count $EU(G)$, not enumerate $eu(G)$. If two traversals reach the same state, i.e., the same current vertex with the same used-edge list in the middle of the traversals, the valid traversals of unused edges after the state will also be the same. For example, on the graph in Figure 4.3, two traversals $(1,2,3)$ and $(3,2,1)$ reach the same current vertex b with the same used-edge list $\{1,2,3\}$. In fact, the two traversal have only the same valid traversal $(4,6,5)$ after this state. This indicates that if a traversal reaches the same state reached in a previous traversal, the number of valid traversal after the state is the same as previous one. Thus we can reduce duplicated traversals by memorizing the number of valid traversals after each state in the middle of traversals.

This algorithm can be considered as dynamic programming with the following recursion:

$$EU(v,E') = \sum_{e=\{v,u\}\in E'} EU(u,E' \setminus \{e\}),$$

where E' stands for a unused-edge sets, and set $EU(w,\Phi) = 1$ for all $w \in V$. Then we obtain $EU(G) = EU(v_1,E)$. The space and time complexity of this algorithm is bounded by $O(n2^m)$ and $O(m2^m)$, respectively: Since the states is defined as a pair of a vertex and a used-edge list, the number of states is bounded by the product of the number n of vertices and the number 2^m of the valid used-edge list. The space required by this algorithm is bounded by the number of states, i.e., $O(n2^m)$. The number of steps of this algorithm is bounded by the sum of the number of valid edge-selection at each state. For each vertex v, the upper bound of the edge-selection is its degree $d(v)$. Hence the number of steps is bounded by $O(m2^m)$. This is exponential improvement from the complexity $O(m \cdot m!)$ of the naïve backtracking.

connected. This means $n-1 \leq m$ and hence $n+m = O(m)$.

4.2.3 Proposed Method

We aim to construct the Rot-πDD representing the set of edge-permutations that correspond to Eulerian trails. We first try to design an algorithm to construct the πDD representing Eulerian trails based on dynamic programming, and observe that dynamic programming cannot be directly applied to construction of the πDD. Next we notice that a property of left-rotation operations suits dynamic programming based approach. We thus introduce a Rot-πDD construction algorithm based on dynamic programming using the property.

To construct the πDD representing Eulerian trails, we utilize a property that a permutation can also represent a state in the middle of a traversal. We assume a state at which we already use k edges. Then, we use a permutation and k to represent the state: first k elements are used edges in the used order, and remaining $m-k$ elements are unused edges. When $k=m$, a permutation represents an Eulerian trail. For example, if we traverse edges $(3,2,1,4)$ in this order on the graph in Figure 4.3, this state is represented by $(3,2,1,4,5,6)$ with $k=4$, for example. Since we fix a start vertex as v_1, we can determine the vertex at the second step from the first used-edge, and then we can determine the vertex at the third step, and so on. Hence we can also determine the current vertex from the used-edge order and k. Furthermore, we can use a transposition to add an edge to the used-edge list in permutation representation. In the above example, we assume to use edge 6 in next step of a traversal. Then we should move 6, at the 6-th position, to the 5-th position. Hence transposition $\tau_{5,6}$ realizes this move, and then the new state is represented by $(3,2,1,4,6,5)$ with $k=5$. More formally, using the edge with id x at the k-th step can be realized by a transposition $\tau_{i,k}$, where i is the position of element x in the permutation representing a state.

We use this permutation representation of states in backtracking algorithm, and simultaneously construct πDDs in top-down manner. In a traversal with a state permutation π and the number k of steps, if we use an edge and the usage corresponds to a transposition $\tau_{x,y}$, we recursively construct the πDD P for the state $\tau_{x,y}\cdot\pi$ with $k+1$ and let the 1-edge from the node with $\tau_{x,y}$ point to P, and let the 0-edge from the node point to the πDDs corresponding to using other edges incident to the current vertex. We notice that, hereafter, we reverse the order of permutation representation because transpositions in πDDs are ordered in decreasing order from the top to the bottom, which means we should determine the elements in a permutation from the right to the left in traversals. For example, $(6,5,4,1,2,3)$ with $k=4$ represents the used-edge list $(3,2,1,4)$ in the used order, and unused-edges 5 and 6.

If two states in different traversals have the same $(m-k)$-prefix of permutation representations and the same k, the same πDD represents the valid traversals after the states. Therefore we can share the πDD nodes for such equivalent states. However, this is less

compressed than dynamic programming approach for counting: even if two permutations have the same unused-edge list, the order of the unused edges in the permutations can be different. For example, two states $(6,5,4,3,2,1)$ with $k=3$ and $(4,5,6,1,2,3)$ with $k=3$ are yielded by the traversals $(1,2,3)$ and $(3,2,1)$, respectively, and have the same unused-edges and the same current vertex. However they are not shared in a πDD because the order of unused edges are different in their permutations. This indicates the compression by πDDs and construction time will be properly worse than dynamic programming.

Instead of transpositions, we can also use left-rotations to represent using edges: using an edge with id x at the k-th step can be realized by a left-rotation $\rho_{i,k}$, where i is the position of the element x in the permutation representing a state. For example, we assume that we want to use edge 5 on the state represented by $(4,5,6,1,2,3)$ with $k=3$, Then we use left-rotation $\rho_{2,3}$, and obtain the new permutation representation $(4,6,5,1,2,3)$. We notice that the first k elements in a permutation after left-rotations from m to $k+1$ must be in the increasing order because $\rho_{i,j}$ must not change the relative order of the first $j-1$ elements. This means unused edges in a state must be in increasing order in its permutation representation. Thus, if two states have the same unused-edge list, the $(m-k)$-prefixes of their permutation representations are also the same. Hence using left-rotations makes Rot-πDDs be compact as well as dynamic programming.

Algorithm 4.2.1 show the pseudo code of our Rot-πDD construction algorithm. We call EulerianTrail(G, v_1, ι, 0) to construct the Rot-πDD representing all Eulerian Trails in a graph G. The time complexity of our algorithm is $O(m2^m)$, which is same as dynamic programming. On the other hand, he size of a resulting Rot-πDD is bounded by $O(m2^m)$, which is worse than dynamic programming; binary branching of Rot-πDDs requires us to make at most $d(v)$ Rot-πDD nodes for edge-selection from a vertex v.

4.2.4 Computational Experiments

We conducted computational experiments to evaluate the practical performance of our algorithms. We implemented our algorithm in C++ with gcc 4.9.3 complier. We carried out experiments on a 3.20 GHz CPU machine with 64 GB memory. We use two types of graphs: complete graphs and random graphs. The complete graph with n vertices, denoted by K_n, is a graph with $n(n-1)/2$ edges for all the pair of two vertices. A random graph with n vertices and m edges is generated by a random walk, which repeats to select a next vertex randomly and add an edge connects the current vertex and the next vertex.

A complete graph K_n for $n \geq 3$ is Eulerian if and only if n is odd. Hence we use K_n's for $n = 3,5,7,9$. Table 4.1 shows the experimental results for complete graphs.

Algorithm 4.2.1 Algorithm to construct a Rot-πDD representing all Eulerian trails.

```
1:  procedure EULERIANTRAILS(G, v, π, k)
2:      if k = m then
3:          return the 1-sink
4:      else if have never memorized the Rot-πDD ℙ for (m−k)-prefix of π then
5:          ℙ ← the 0-sink
6:          for i from m−k to 1 do
7:              if an endpoint of edge π_i is v then
8:                  u ← another endpoint of π_i
9:                  ℙ ← (ρ_{i,m−k}, ℙ, EULERIANTRAIL(G, u, π·ρ_{i,m−k}, k+1))
10:             end if
11:         end for
12:     end if
13:     return ℙ
14: end procedure
```

Table 4.1. The experimental results for complete graphs K_n with odd n

n	m	EU(G)	the size of Rot-πDD	runtime (sec)
3	3	2	2	0.00
5	10	528	284	0.00
7	21	389928960	422988	0.55
9	36	-	-	$>$ 1000.00

Since the growth of the number of edges in a complete graph is quadratic to the number of vertices, $EU(K_n)$ and calculation time dramatically increase. This result follows the complexity analysis of our algorithm. In $n = 9$ case, the algorithm cannot finish in the time limit 1000 seconds. This requires us to improve the performance if we aim to handle larger dense graphs. On the other hand, $n = 7$ case shows our algorithm achieve to compress 400 million Eulerian trails into a Rot-πDD with 400 thousand nodes.

Table 4.2 shows the experimental results for random graphs. Compared to complete graphs, our algorithm runs faster for random graphs even if its vertex set and edge set are larger. It may be because in practice, the number of valid traversals on random graphs is less than one on complete graphs. Our algorithm achieves to construct a Rot-πDD for an $m = 36$ random graph in about 3 minutes, while it fails for $m = 36$ complete graph K_9. The compression is extremely high; the compression ratio, namely EU(G) divided by the size of a Rot-πDD, is about 680,000. The results for random graphs indicates that our algorithm will run faster than the estimation from the theoretical complexity.

Table 4.2. The experimental results for random graphs

n	m	EU(G)	the size of Rot-πDD	runtime (sec)
8	11	4	21	0.00
9	14	576	286	0.00
10	18	20736	1894	0.00
11	22	1990656	18726	0.01
12	26	240537600	30534	0.11
13	31	24365629440	1150285	3.55
14	36	32985223004160	48657136	182.04
15	42	-	-	$>$ 1000.00

4.2.5 Concluding Remarks for Eulerian Trail Enumeration

In this section, we provide an algorithm to enumerate Eulerian trails in the form of a Rot-πDD. We theoretically analyze the complexity of the Rot-πDD construction algorithm and conclude that Rot-πDDs are preferable to πDDs.

Application to practical problems such as DNA fragment assembly and circuit design is one of future directions. In addition, we would like to consider algorithms for enumeration problems of generalized version of Eulerian trails: we can ignore at most k edges or use at most k edges twice for fixed k.

Furthermore, our technique, construction of a Rot-πDD based on dynamic programming, can be considered as a framework for several ordering problems on a graph. For instance, Hamiltonian paths, which is defined the path passing through every vertex exactly once, is solved similar dynamic programming for the vertex set of a given graph, instead of the edge set. Hence a Rot-πDD for Hamiltonian paths is also obtained by an adjusted version of our algorithm. Topological ordering problem in the next section is also an instance of such problem, and we propose an algorithm for the problem based on dynamic programming approach too. It is important future work to reveal and formalize what kind of problems are solved with this framework.

4.3 Enumeration of Topological Orders

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 several graph algorithms [22]. 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 in 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 [70].

Linear time algorithms calculating a topological order are classical and well-known algorithms, and dealt with by Cormen et al. [22]. 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. [8] proposed a topological sort algorithm which allows edge insertions, and Pearce et al. [65] proposed an algorithm which can also handle edge deletions. Another problem is the enumeration problem of all topological orders. Ono et al. [64] 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. [15] proved that it is $\#P$-complete. Bubley et al. [18] proposed a randomized algorithm to approximate the number of all linear extensions. Li et al. [53] provided an experimentally fast algorithm counting all topological orders based on Divide & Conquer technique. 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 [3, 21].

In this section, 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 as large graph as possible.

The proposed method is based on πDDs. Although a πDD can be used to achieve our purpose, compression ratio and query processing are not efficient enough practically or theoretically. Thus, we use Rot-πDDs. 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, and the theoretical complexity is also bounded.

4.3.1 Contribution Summary

Contributions for this problem are summarized as follows.

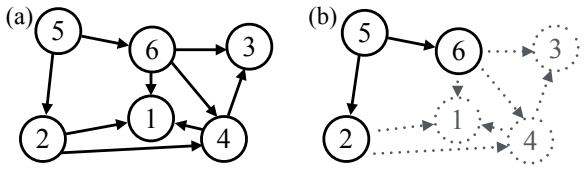


Figure 4.4. (a) A DAG and (b) the subgraph induced by the vertex set $\{2,5,6\}$.

- 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 in general, it is difficult to estimate the size and computation time of decision diagrams.

Experimental results, which will be described later, show that the proposed algorithm and data structure work very well: 3.7×10^{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.

4.3.2 Topological Orders

We first define *topological orders* of a directed graph, which are our objectives in this section.

Definition 4.3.1. *A* topological order *of a directed graph G is an ordering $(v_1, v_2, \ldots, v_n)$ of all vertices such that v_i must precede v_j if $(v_i, v_j) \in E$.*

Example 4.3.1. *The graph in Figure 4.4(a) has four topological orders:* $(5,2,6,4,1,3)$, $(5,2,6,4,3,1)$, $(5,6,2,4,1,3)$, *and* $(5,6,2,4,3,1)$.

In this section, we assume that given graphs are DAGs because we can determine whether or not a directed 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 [44, 78]. 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. 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 G. Since $TO(G) = O(n!)$, the time complexity of the worst case is $O((n+m)n!)$. We can improve this complexity by a dynamic programming (DP) approach.

For example, in Figure 4.4(a), we can delete vertices $\{1,3,4\}$ in the order $(1,3,4)$ or $(3,1,4)$, where we 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 $(1,3,4)$ and $(3,1,4)$. 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 DP in top-down manner, which recursively calculates $TO(G[V']) = \sum_{v \in V', d_-(v)=0} TO(G[V' \setminus \{v\}])$. 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 the 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 the 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 [21]. 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 4.3.5.

Here, we remember our goal in this section 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 $TO(G)$.

4.3.3 Decision Diagram Construction Method

In this section, we first discuss whether or not compression of a πDD is suitable for the DP approach. As a result, it is no; a similar problem for enumeration of Eulerian trails in Section 4.2 appears. Thus we consider to use Rot-πDDs for DP and provide a construction algorithm.

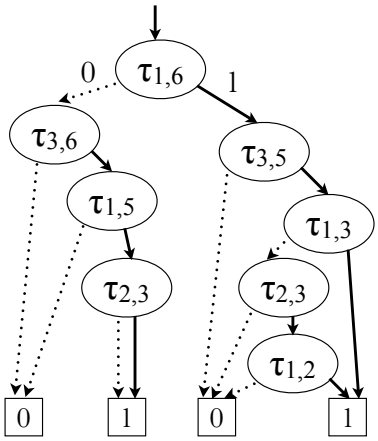


Figure 4.5. The πDD representing $\{(5,2,6,4,1,3),(5,2,6,4,3,1),(5,6,2,4,1,3),(5,6,2,4,3,1)\}$.

DP Approach and πDDs

Now, we consider whether or not we can directly construct a πDD in the same way as the DP approach described in Section 4.3.2.

Here, we note that transposition decomposition 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, like edge set for Eulerian trails in Section 4.2.3. Furthermore, a deletion of a vertex v can be described as a transposition $\tau_{i,k}$, where i is the position of v in the permutation. For example, we can consider a permutation (6, 2, 5, 4, 3, 1) represents the subgraph in Figure 4.4(b) such that the deletion order is $(1,3,4)$. 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 $(5,2,6,4,3,1)$, which represents the subgraph induced by $\{2,5\}$ and the reverse order of deletions.

By compressing swap sequences into a πDD, we can recursively construct a πDD for all topological orders. That is, for each recursion represented as a permutation π, if we apply $\tau_{i,j}$ to delete π_i, we create the new πDD such that its root node is $\tau_{i,j}$, its 1-edge child is the πDD for transposition sequences of the induced subgraph yielded by deleting π_i, and its 0-edge child is the πDD for transposition sequences without deleting π_i at the $(n-j)$-th step. The πDDs for the 1-edge and 0-edge child are recursively constructed. Figure 4.5 illustrates an example of a πDD representing topological orders of the graph in Figure 4.4(a).

However, deletions by transpositions are not available for DP. In order to use DP approach, transposition sequences for the same induced subgraph must be uniquely determined. Even if different prefixes of permutations represent the same induced sub-

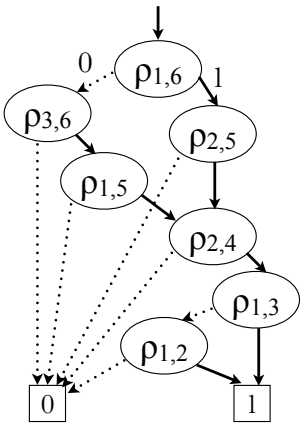


Figure 4.6. The Rot-πDD representing $\{(5,2,6,4,1,3),(5,2,6,4,3,1),(5,6,2,4,1,3),(5,6,2,4,3,1)\}$.

graph, their transposition sequences can differ. For example, consider the DAG in Figure. 4.4. Deletion sequences $(3,1,4)$ and $(1,3,4)$ generate the same induced subgraph on $\{2,5,6\}$, and these states are represented as $(5,2,6,4,1,3)$ and $(6,2,5,4,3,1)$, respectively. The induced subgraph on $\{2,5,6\}$ has a topological order $(5,2,6)$. In order to obtain this, we apply no transposition to $(5,2,6,4,1,3)$, while we apply $\tau_{1,3}$ to $(6,2,5,4,3,1)$. This means there are multiple πDDs corresponding to the same induced subgraph.

DP Approach and Rot-πDDs

As described in the previous subsection, the DP approach cannot be used to directly construct a πDD. To overcome this problem, we use a left-rotation decomposition of a permutation. 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 ith 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 ι and apply $\rho_{i,j}$ in decreasing order of j.

Thus, we can use the DP approach by using Rot-πDDs Figure 4.6 illustrates the Rot-πDD for the same set as Figure 4.5. We can see that topological orders of the induced subgraph $G[\{2,5,6\}]$ yielded by deletion sequences $(3,1,4)$ and $(1,3,4)$ are shared as the sub-Rot-πDD with a root node $\rho_{1,3}$, whereas the πDD fails to share them. Algorithm 4.3.1 describes the DP based construction algorithm of a Rot-πDD.

4.3.4 Rot-πDD Operation for Dynamic Edge Addition

Some queries such as random samplings and counting the cardinality of the set represented by a Rot-πDD are available to analyze topological orders in a Rot-πDD. The

Algorithm 4.3.1 Rot-πDD construction for all topological orders of $G = (V, E)$.

```
1: procedure ConstructRotPiDD(G)
2:     if V is empty then
3:         return 1-sink
4:     else if have never memorized the Rot-πDD P_G for G then
5:         P_G ← the 0-sink
6:         for each v whose out-degree is 0 in G do
7:             i ← v's position in the increasing sequence of V, j ← |V|
8:             P_G ← (ρ_{i,j}, P_G, ConstructRotPiDD(G[V \ {v}]))
9:         end for
10:    end if
11:    return P_G
12: end procedure
```

runtime of these operations depends on only the size of the Rot-πDDs by using memo cache techniques.

On the other hand, some queries have to be newly designed. For example, the precedence query Precede$(\mathbb{P}, u, v)$ returns the Rot-πDD that represents only permutations π extracted from the Rot-πDD $\mathbb{P}$ such that u precedes v in π. This query is equivalent to addition of the edge (u, v) to a DAG. This query can be designed as a recursive procedure described in Algorithm 4.3.2.

The idea of the algorithm is simulation of moves of the two elements u and v. Initially, we start with the identity permutation, i.e. u and v are at the u-th position and the v-th position, respectively. After a rotation, the positions of u and v may be changed. If u or v are out of the range of later rotations, their relative order is fixed and we can check whether or not u precedes v. The runtime of a precedence query also depends on only the size of the Rot-πDDs thanks to memo cache.

4.3.5 Theoretical Analysis

In this section, we analyze the time and the space complexity of DP based counting and Rot-πDD 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 upper bound $O(n^w)$ of $IS(G)$. According to Dilworth's theorem [25], 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.

Algorithm 4.3.2 Precedence query for a Rot-πDD $\mathbb{P}$.

1: **procedure** PRECEDE($\mathbb{P}, u, v$)
2: **if** $\mathbb{P}$ is the 0-sink **then**
3: **return** the 0-sink
4: **else if** $\mathbb{P}$ is the 1-sink **then**
5: **if** $u < v$ **then**
6: **return** the 1-sink
7: **else**
8: **return** the 0-sink
9: **end if**
10: **end if**
11: **if** the result of PRECEDE($\mathbb{P}, u, v$) is memorized on cache **then**
12: **return** the memorized result
13: **end if**
14: set x and y such that the root node of $\mathbb{P}$ is $\rho_{x,y}$.
15: **if** $y < u$ and $v < u$ **then** ▷ if u will not move and u is on the right of v
16: **return** the 0-sink
17: **else if** $y < v$ and $u < v$ **then** ▷ if v will not move and v is on the right of u
18: **return** $\mathbb{P}$
19: **end if**
20: $\mathbb{P}_0 \leftarrow$ the left child of $\mathbb{P}$, $\mathbb{P}_1 \leftarrow$ the right child of $\mathbb{P}$
21: $nu \leftarrow u, nv \leftarrow v$
22: **if** $x = u$ **then**
23: $nu \leftarrow y$ ▷ $\rho_{x,y}$ moves u at the y-th position
24: **else if** $x < u$ **then** ▷ Note that Line 15 ensures $u \leq y$
25: $nu \leftarrow u - 1$ ▷ $\rho_{x,y}$ moves u at the $(u-1)$-th position
26: **end if**
27: **if** $x = v$ **then**
28: $nv \leftarrow y$ ▷ $\rho_{x,y}$ moves v at the y-th position
29: **else if** $x < v$ **then** ▷ Note that Line 17 ensures $v \leq y$
30: $nv \leftarrow v - 1$ ▷ $\rho_{x,y}$ moves v at the $(v-1)$-th position
31: **end if**
32: Memorize ($\rho_{x,y}$, PRECEDE($\mathbb{P}_0, u, v$), PRECEDE($\mathbb{P}_1, nu, nv$)) on cache
33: **return** ($\rho_{x,y}$, PRECEDE($\mathbb{P}_0, u, v$), PRECEDE($\mathbb{P}_1, nu, nv$))
34: **end procedure**

Theorem 4.3.1. *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 $l_i \leq n$, but in fact $\sum_{k=1}^{w} l_k = n$ holds. We can prove a tighter bound using this restriction.

Lemma 4.3.1. *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 done inductively over w.

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

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

Here, $\displaystyle\sum_{k=1}^{x+1} l_k = n$ implies $\displaystyle\sum_{k=1}^{x} l_k = n - l_{x+1}$. 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 $1\leq a\leq n$ and $2\leq x$, $f(a)$ is upward-convex if $a<\frac{2n}{x+1}+1$, while $f(a)$ is downward-convex otherwise. Because $\frac{n}{x+1}<\frac{2n}{x+1}+1\leq n+x$ holds, we can conclude $f(\frac{n}{x+1})=(\frac{n}{x+1}+1)^{x+1}$ is the maximum of $f(a)$ for all $1\leq a\leq n$, and

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

holds. This proves the induction step. □

Corollary 4.3.1. *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 4.3.1 and Lemma 4.3.1. □

Corollary 4.3.1 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-πDD representing all topological orders and the time of the construction. The size of such a Rot-πDD 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-πDD has exactly two edges. Therefore, the size of such a Rot-πDD is at most $O(w(n/w+1)^w)$[4]. 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-πDD representing all topological orders is $O((n+m)(n/w+1)^w)$.

4.3.6 Computational Experiments

We measured the performance of our Rot-πDD 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 πDD construction, DP counting, and Divide & Conquer counting [53]. Since direct construction of a πDD seems to be 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.

Figure 4.7 and Figure 4.8 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 4.3 shows the average numbers of topological orders, the sizes of Rot-πDDs, and runtimes on the case $k = 1$. It shows the amazing efficiency of Rot-πDDs: 3.7×10^{41} topological orders are compressed into a

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

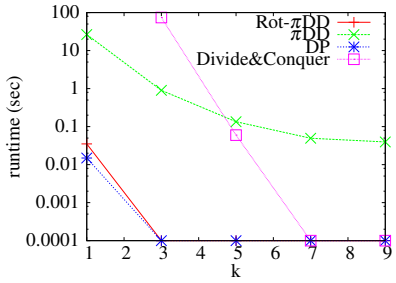


Figure 4.7. Average runtime for construction when $n = 20$.

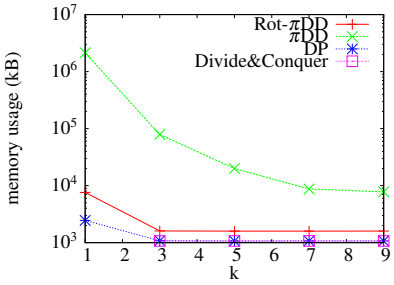


Figure 4.8. Average memory usage for construction when $n = 20$.

Table 4.3. Experimental results on the cases $k = 1$.

n	The number of topological orders	Rot-πDD 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	188972608055858740408591893 98	2634015	3.78
40	19224622437706527112568934998 0187	4649639	6.68
45	75068589270080843845910704526224 56252	8288752	12.69
50	37563660779499151811427427955995 2431497225	22542071	35.51

Rot-πDD that has only 2.2×10^7 nodes in 36 seconds on the case $n = 50$. Note that each node of Rot-πDDs consumes about 30 bytes.

Figure 4.9 and Figure 4.10 show the average runtime and memory usage on $k = 1$ cases. πDD and Divide & Conquer time-out on the case $n \geq 25$ and $n \geq 20$, respectively. We can obtain a Rot-πDD, 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-πDD into the hash table, and the overhead memory is caused by the difference of the space complexities between DP and Rot-πDD as described in Section 4.3.5.

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-πDD 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-πDD and a DP table. Note that the DP table size equals $IS(G)$.

Figure 4.11 and Figure 4.12 show the results for query processing. In almost all cases, Rot-πDDs can generate and index all topological orders faster than or equal to

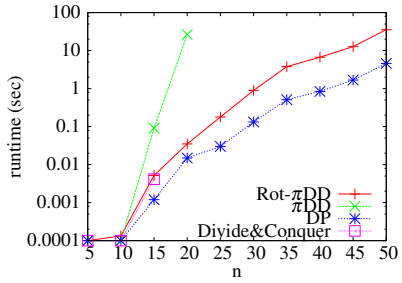


Figure 4.9. Average runtime for construction when $k = 1$.

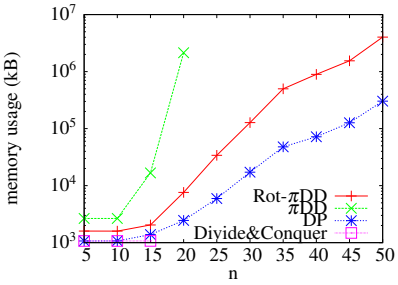


Figure 4.10. Average memory usage for construction when $k = 1$.

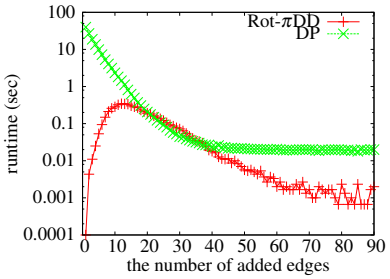


Figure 4.11. Average runtime for edge addition queries.

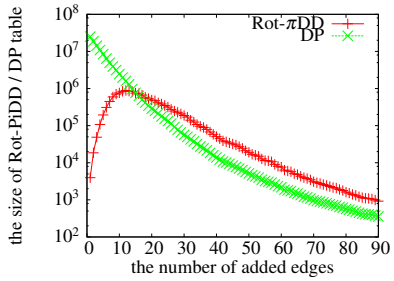


Figure 4.12. Average space for edge addition queries.

DP. Especially in sparse cases, query processing of Rot-πDDs is very efficient. It may be because Rot-πDDs (and πDDs) can represent the set of all n-permutations with $n(n-1)/2$ nodes.

4.3.7 Concluding Remarks for Topological Orders

In this section, we gave an efficient method for generating and indexing all topological orders of a given DAG. We showed that Rot-πDDs are suitable for indexing topological orders in terms of both of theory and practice. In addition, we proposed a query algorithm for dynamic edge addition to a DAG and it efficiently works on our computational experiments.

Future work is to apply Rot-πDDs 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-πDD construction technique to other graph generation problems which can be solved by considering induced subgraphs recursively, e.g. perfect elimination orderings [56].

4.4 Enumeration of Pattern-avoiding Permutations

A permutation π *avoids* a *pattern* σ if no subsequence in π is order isomorphic to σ. Two numerical sequences $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_m)$ are *order isomorphic* if a and b have the same length and satisfy the rule $a_i < a_j$ if and only if $b_i < b_j$ for all i, j. Permutations that avoid pattern σ are called σ-*avoiding* permutations.

Research of pattern-avoiding permutations dates back to *stack sort*, which was proposed by Knuth in [48]. In stack sort, we can use a single stack to sort elements. Knuth showed that a permutation is stack sortable if and only if it is a 231-avoiding permutation. Several variations of the stack sorting problem, such as the twice stack sorting problem [81] and the double-ended queue sorting problem [69], have been proposed, and pattern-avoiding permutations were developed in that context.

After pattern-avoiding permutations were proposed, many researchers were engaged in studies to compute the number of the permutations that avoid given patterns. For example, 1342-avoiding permutations have been enumerated by a mathematical approach [12], and 1324-avoiding permutations can be counted by computer programs [54]. Moreover, the relation of classes on pattern-avoiding permutations has also been examined. Two permutations π and σ are *Wilf-equivalent* if the number of n-permutations avoiding π is equals to the number of σ-avoiding n-permutations for all positive integers n. In [76], the nontrivial Wilf-equivalence between the 4132-avoiding permutations and the 3142-avoiding permutations was discovered. The generation of pattern-avoiding permutations can contribute to not only the discovery of unknown Wilf-equivalent classes, but also the identification of bijective functions between such classes.

Relations between pattern-avoiding permutations and mathematical problems have been studied actively [19, 33]. In particular, Yao et al. [88] revealed a bijection between *mosaic floorplans* and *Baxter permutations*, which are generalized pattern-avoiding permutations, and Ackerman et al. proposed a simple encoding and decoding between them in [1]. A floorplan is a topological partition of a rectangle into multiple rectangles, and a mosaic floorplan is a subclass of floorplans. Floorplans have practical applications in areas such as VLSI design [37]. Storing all pattern-avoiding permutations into a database is equivalent to preparing a database of floorplans. Database queries such as searching by criteria and random sampling are useful for VLSI design. Therefore, generating pattern-avoiding permutations can contribute to solving practical problems.

Wilf provided an amortized polynomial-delay algorithm that generates all permutations avoiding the identity pattern, and posed the question about the complexity of generation for other patterns [82]. In [13], Bose et al. proved that the general counting problem is #P-complete. Generating algorithms for some particular patterns were proposed [28]. For practically fast enumeration, PermLab [2] has been developed by Albert. This is a software to analyze permutation patterns in several aspects, and has

a function to enumerate pattern-avoiding permutations. Therefore, fast enumeration algorithms for pattern-avoiding permutations are desired in both of practical applications and theoretical researches.

4.4.1 Contribution Summary

We provide a practically efficient algorithm for implicitly generating pattern-avoiding permutations with πDDs. Furthermore, we extend this algorithm to handle some generalized patterns, such as vincular patterns and bivincular patterns. This algorithm runs practically faster than naïve listing algorithm and less space than naïve array representation.

Furthermore, we use Rot-πDDs for the previous algorithm instead of πDDs. We prove that the sizes of Rot-πDDs used in the middle of algorithm are theoretically bounded by $O(n^2)$, while the sizes of corresponding πDDs seems to be difficult to estimate and experimentally larger than the sizes of Rot-πDDs. Experimental results shows that Rot-πDDs indeed accelerates the whole time of the algorithm.

4.4.2 Permutation Patterns

We first define containment and avoidance of permutation patterns.

Definition 4.4.1. *Two numerical sequences $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_m)$ are* order isomorphic *if a and b have the same length and $a_i < a_j$ if and only if $b_i < b_j$ for all $1 \leq i, j \leq n$.*

Definition 4.4.2. *A permutation π* contains *a permutation σ if there is at least one subsequence in π which is order isomorphic to σ, where the subsequence need not consist of consecutive numbers in π. Such σ is called a (permutation)* pattern *and π is called a* text. *Conversely, π* avoids *σ if π does not contain σ, and π is a σ-avoiding permutation. We may briefly write a pattern in the one line form without parentheses and commas.*

Example 4.4.1. *A permutation $(4,2,1,3)$ contains a pattern $(3,1,2)$ because $(4,2,3)$ and $(4,1,3)$ are order isomorphic to the pattern. Thus, $(4,2,1,3)$ is a 312-avoiding permutation.*

The patterns defined above are also called *classical patterns* because some generalizations have been proposed. *Vincular patterns*, which are also called *generalized patterns*, are one of well-known generalizations [4].

Definition 4.4.3. *Vincular patterns requires not only order isomorphism, but also adjacency of some positions in the permutation. If a vincular pattern requires that the i-th and the $(i+1)$-th elements are adjacent, the corresponding elements in the text must be adjacent. We use the underline notation to represent adjacencies: If the ith and the $(i+1)$th elements are consecutively underlined, they must be adjacent.*

Example 4.4.2. *For example, we consider the permutation $(4,2,1,3)$ and the vincular pattern $3\underline{12}$. Both $(4,2,3)$ and $(4,1,3)$ are order isomorphic to 312, but $(4,2,3)$ does not match $3\underline{12}$ because the second and third elements are not adjacent in the text. In contrast, $(4,1,3)$ matches the pattern because 1 and 3 are adjacent in $(4,2,1,3)$. Thus, $(4,2,1,3)$ contains $3\underline{12}$.*

Vincular patterns have been further extended to *bivincular patterns* [14].

Definition 4.4.4. *A bivincular pattern is restricted by restrictions same as vincular patterns and additionally by consecutiveness of element values. If a bivincular pattern requires that the i-th and the $(i+1)$-th values are consecutive, the corresponding elements in the text must differ exactly by one. We use the two-line form with bars and underlines to represent bivincular patterns. The first row represents consecutiveness and the identity permutation ι, and the second row represents adjacencies and a relative order.*

Example 4.4.3. *A bivincular pattern $\begin{smallmatrix}\overline{12}3\\ 3\underline{12}\end{smallmatrix}$ represents a pattern where the 1st and the 2nd smallest values must be consecutive, the 2nd and the 3rd elements in a subsequence must be adjacent in a text, and the relative order must match 312. Thus, the permutation $(4,2,1,3)$ avoids $\begin{smallmatrix}\overline{12}3\\ 3\underline{12}\end{smallmatrix}$. Indeed both subsequences $(4,2,3)$ and $(4,1,3)$ are order isomorphic to 312 but $(4,2,3)$ does not match the bivincular pattern because 2 and 3 are not adjacent in the text, and $(4,1,3)$ does not match the bivincular pattern either because 1 and 3 are not consecutive.*

The problem considered in this section can be stated as follows: for given a positive integer n and a pattern σ, generate all σ-avoiding permutations of length n. Hereafter, unless otherwise noted, we denote the length of a given pattern σ by k.

4.4.3 πDD-based Method

In this section, we review the previous algorithm for generating all σ-avoiding n-permutations with πDDs. This algorithm makes use of the following fact: the set of σ-avoiding permutations is the complement of the set of permutations that contain σ. Hereafter, $Av_n(\sigma)$ denotes the set of σ-avoiding n-permutations and $C_n(\sigma)$ denotes the set of n-permutations that contain σ. As stated above, $Av_n(\sigma) = S_n \setminus C_n(\sigma)$ holds. In

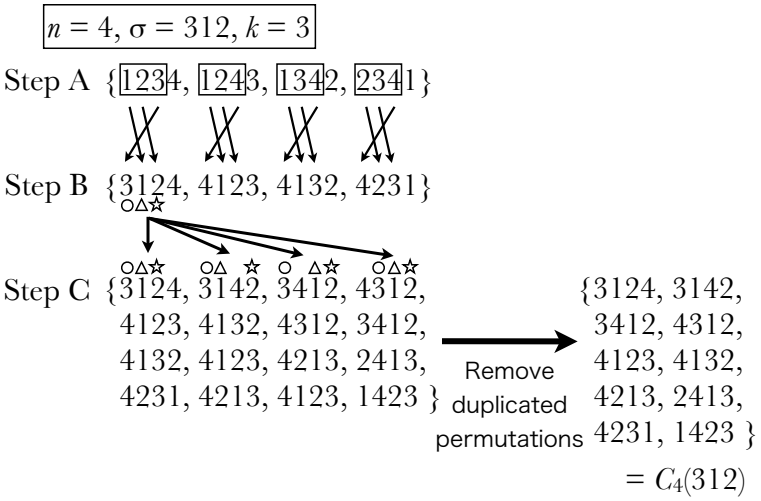


Figure 4.13. The process of generating $C_4(312)$.

general, the time to compute set difference depends on the cardinalities of the sets. On the other hand, the set difference operation of πDD can be efficient because it depends on the size of the πDDs.

We already have the algorithm for construction of $\mathbb{S}_n$. We provide the algorithm for generating $C_n(\sigma)$ for classical patterns. In order to generate $C_n(\sigma)$, we must generate all permutations which have at least one subsequence order isomorphic to σ. This is achieved in three steps as follows.

A. Generate all permutations whose k-prefix is ordered in increasing order.

B. Rearrange the k-prefix of each permutation which was generated in step A so that the k-prefix becomes order isomorphic to σ.

C. Distribute the k-prefix of each permutation π which was generated in step B over $\binom{n}{k}$ possible positions in π.

Figure 4.13 shows the process of generating $C_4(312)$. Step A generates all $\binom{n}{k}$ combinations in the k-prefix of the permutations. Step B rearranges the k-prefix of each permutation into the numerical sequence order isomorphic to σ. All possible numerical sequences order isomorphic to σ appear in the k-prefix of the permutations which are generated in step B. The distribution by step C generates all permutations π such that at least one of $\binom{n}{k}$ subsequences in π exactly matches one of the numerical sequences order isomorphic to σ. Note that they may have some duplications.

Steps B and C involve the rearrangements of multiple permutations. This means that this process can be done by Cartesian products of πDDs as shown in Section 2.3.3. Let $\mathbb{A}$ denote the πDD for permutations which are generated in step A, and let $\mathbb{B}$ and $\mathbb{C}$

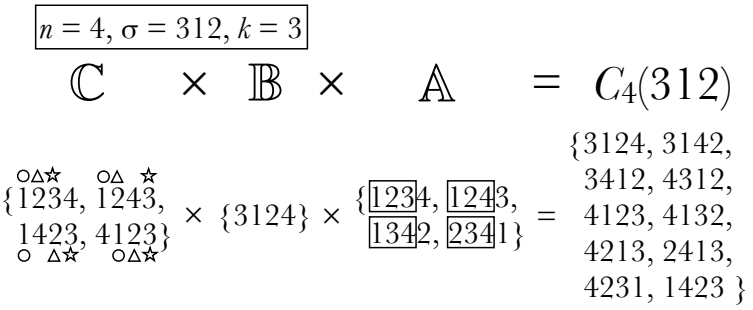


Figure 4.14. Cartesian product for generating $C_4(312)$.

denote the πDDs for the rearrangements which correspond to steps B and C, respectively. Note that the permutations represented in $\mathbb{B}$ are not the permutations obtained after step B by rearranging those in $\mathbb{A}$. The permutations in $\mathbb{B}$ are the permutations as operations to apply those in $\mathbb{A}$. The same applies to $\mathbb{C}$. Then, $C_n(\sigma)$ can be obtained by computing $\mathbb{C} \times \mathbb{B} \times \mathbb{A}$. Figure 4.14 may help to understanding the relation between the process in Figure 4.13 and the Cartesian product. We indeed do not have to consider the duplications since we achieve the process via πDD operations.

We provide the method for the construction of $\mathbb{A}$ at the end because it is similar to the construction of $\mathbb{C}$ but more complicated.

Construction of $\mathbb{B}$

In order to rearrange the k-prefix of all permutations in $\mathbb{A}$ so that it becomes order isomorphic to σ, we define $\mathbb{B}$ to be the πDD consisting only of the permutation σ. To construct this πDD, we first decompose σ into composition of transpositions as given in Definition 2.1.6. The πDD forms one path based on this decomposition, and can be easily constructed in a bottom-up fashion.

Construction of πDD $\mathbb{C}$

We define $\mathbb{C}$ to be the πDD for the set of n-permutations π such that there are k indices $1 \leq p_1 < p_2 < \cdots < p_k \leq n$ with $\pi_{p_i} = i$. This means that each permutation in $\mathbb{C}$ must have the numerical sequence $(1,2,\ldots,k)$ as its subsequence. There is a simple method to construct $\mathbb{C}$. First, for each n-permutation which satisfies the above condition, we construct one πDD like the construction of $\mathbb{B}$. And then, we take the union of the πDDs. This algorithm is simple and easy to implement. However, this is not efficient because this algorithm has to repeat constructions and union operations $\binom{n}{k}$ times.

An idea to reduce the number of πDD operations is based on Pascal's triangle, in which the recursion $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ holds. Let $Pos_{i,j}$ be the set of all n-permutations

Algorithm 4.4.1 Construct πDD $\mathbb{C}$.

```
1: procedure CONSTRUCTC(n,k)
2:     P_{i,j} ← the 1-sink for j = 0, otherwise the 0-sink
3:     for j = 1 to k do
4:         for i = j to n do
5:             P_{i,j} ← P_{i-1,j} ∪ P_{i-1,j-1}.Swap(j,i)
6:         end for
7:     end for
8:     return P_{n,k}
9: end procedure
```

π containing at least one subsequence $\pi_{k_1}\pi_{k_2}\ldots\pi_{k_j}$ satisfying the following two conditions:

1. $1 \leq k_1 < k_2 < \ldots < k_j \leq i$,
2. $\pi_{k_1}\pi_{k_2}\ldots\pi_{k_j} = 12\ldots j$.

It is obvious that $\mathbb{C}$ is the πDD for $Pos_{n,k}$. If we can calculate $Pos_{i,j}$ using $Pos_{i-1,j}$ and $Pos_{i-1,j-1}$ like Pascal's triangle, we can obtain $\mathbb{C}$ with only $O(kn)$ operations. In order to make this idea work, we restrict $Pos_{i,j}$ with the additional condition as follows:

3. For each $i+1 \leq x \leq n$, x is fixed, i.e., $\pi_x = x$.

Here, we can partition $Pos_{i,j}$ into the two sets: the set including π with $\pi_i \neq j$ and the other set. The former set equals $Pos_{i-1,j}$, and the latter one can be obtained by assigning j to the ith position of permutations in $Pos_{i-1,j-1}$. Hence, this is achieved by $\mathbb{P}_{i-1,j-1}$.Swap(j,i), where $\mathbb{P}_{i,j}$ denotes the πDD for $Pos_{i,j}$, because the ith element is i from the third condition and j is not assigned yet. Thus, $\mathbb{P}_{i,j} = \mathbb{P}_{i-1,j} \cup \mathbb{P}_{i-1,j-1}$.Swap(j,i) holds. The dynamic programming for this recursion is shown in Algorithm 4.4.1.

Construction of πDD $\mathbb{A}$

As stated above, $\mathbb{A}$ is the πDD for the set of all n-permutations whose k-prefix is ordered in increasing order. More precisely, a permutation π in $\mathbb{A}$ satisfies $1 \leq \pi_1 < \pi_2 < \ldots < \pi_k \leq n$.

We can obtain $\mathbb{A}$ by repeating assignments in a similar way to the construction of $\mathbb{C}$. However, if we assign elements in increasing order, it does not work. In order to assign v to the p-th position by Swap(p,v), p must be fixed to the p-th position. But p can be at another position by other swaps before executing Swap(p,v) because $p \leq v$ holds. Otherwise, on the construction of $\mathbb{P}_{p,v}$, there is no problem because after

Swap(p,v) assigns v to the p-th position, there is no assignment to the vth position due to $v \leq p$. Therefore, in order to fix the position of elements which will be used later, we reverse the order of assignments. Let $Inc_{i,j}$ be the set of all n-permutations satisfying the following conditions:

1. $i+1 \leq \pi_{j+1} < \pi_{j+2} < \ldots < \pi_k \leq n$,
2. for each $1 \leq x \leq j$, x is fixed, i.e., $\pi_x = x$.

These conditions mean that for each permutation π in $Inc_{i,j}$, the $(k-j)$-suffix of the k-prefix of π is already used and the j-prefix of the k-prefix of π is fixed. We execute the constructions from $Inc_{n,k} = \{\iota\}$ to $Inc_{0,0}$. Let $\mathbb{I}_{i,j}$ denote the πDD for $Inc_{i,j}$. Here, $\mathbb{I}_{i,j} = \mathbb{I}_{i+1,j} \cup \mathbb{I}_{i+1,j+1}$.Swap$(j+1,i+1)$ holds because $\mathbb{I}_{i,j}$ can be partitioned into the set including π with $\pi_{j+1} = i+1$, which is $\mathbb{I}_{i+1,j+1}$.Swap$(j+1,i+1)$ and the other set, which is $\mathbb{I}_{i+1,j}$, like $\mathbb{P}_{i,j}$.

The construction of $\mathbb{A}$ is not completed yet because the $(n-k)$-suffix of each permutation in $\mathbb{I}_{0,0}$ is in one fixed order. We must generate all orders of the $(n-k)$-suffix of each permutation in $\mathbb{I}_{0,0}$. It is realized by $\mathbb{S}_{n,k} \times \mathbb{I}_{0,0}$, where $\mathbb{S}_{n,k}$ is the πDD for the set including the n-permutations π in which $\pi_i = i$ holds for $1 \leq i \leq k$ and the $(n-k)$-suffix is in any order. We can obtain $\mathbb{S}_{n,k}$ by the construction like Algorithm 2.3.1 for $\mathbb{S}_n$. Algorithm 4.4.2 describes the entire process.

Generating permutations containing a vincular pattern

The additional restriction of vincular patterns is adjacency of positions. Therefore, we can generate vincular pattern-avoiding permutations by a slight modification of step C. We call the modified step C'. We denote by $\mathbb{C}'$ the πDD which corresponds to step C'.

If the j-th and the $(j+1)$-th elements must be adjacent, $j+1$ is the right-hand neighbor of j for all permutations in $\mathbb{C}'$. In other words, if we assign j to the i-th position, we must assign $(j+1)$ to the $(i+1)$-th position. For $\mathbb{C}'$, we define $\mathbb{P}'_{i,j} = \mathbb{P}'_{i-1,j-1}$.Swap$(j,i)$ if the j-th and the $(j+1)$-th elements must be adjacent, and otherwise $\mathbb{P}'_{i,j} = \mathbb{P}'_{i-1,j} \cup \mathbb{P}'_{i-1,j-1}$.Swap$(j,i)$ as $\mathbb{P}_{i,j}$. As shown for Algorithm 4.4.1, $\mathbb{P}'_{i-1,j}$ consists only of permutations π such that $\pi_i \neq j$, and $\mathbb{P}'_{i-1,j-1}$.Swap(j,i) consists only of π such that $\pi_i = j$. Thus, if the j-th and the $(j+1)$-th elements must be adjacent, $\mathbb{P}'_{i,j}$ includes only permutations π such that $\pi_i = j$, and $\mathbb{P}'_{i,j}$.Swap$(j+1,i+1)$ includes only permutations π such that $\pi_i = j$ and $\pi_{i+1} = j+1$, that is, j and $j+1$ are adjacent. Therefore, $\mathbb{P}'_{i+1,j+1} = \mathbb{P}'_{i,j+1} \cup \mathbb{P}'_{i,j}$.Swap$(j+1,i+1)$ includes only the permutations in which j and $j+1$ are adjacent because $\mathbb{P}'_{i,j}$.Swap$(j+1,i+1)$ satisfies the adjacency as above, and $\mathbb{P}'_{i,j+1}$ also satisfies the adjacency recursively. Therefore, we obtain $\mathbb{C}' = \mathbb{P}'_{n,k}$ by adding a branch to Algorithm 4.4.1.

Algorithm 4.4.2 Construct πDD $\mathbb{A}$.

procedure CONSTRUCTA(n,k)
  $\mathbb{I}_{n,k} \leftarrow$ the 1-sink
  for $i = n-1$ to 0 **do**
    for $j = k$ to 0 **do**
      if $j < k$ **then**
        $\mathbb{I}_{i,j} \leftarrow \mathbb{I}_{i+1,j} \cup \mathbb{I}_{i+1,j+1}$.Swap$(j+1, i+1)$
      else
        $\mathbb{I}_{i,j} \leftarrow \mathbb{I}_{i+1,j}$
      end if
    end for
  end for

  $\mathbb{S}_{k,k} \leftarrow$ the 1-sink
  for $i = k+1$ to n **do**
    $\mathbb{S}_{i,k} = \mathbb{S}_{i-1,k}$
    for $j = k+1$ to $i-1$ **do**
      $\mathbb{S}_{i,k} \leftarrow \mathbb{S}_{i,k} \cup \mathbb{S}_{i-1,k}$.Swap$(j, i)$
    end for
  end for
  return $\mathbb{S}_{n,k} \times \mathbb{I}_{0,0}$
end procedure

Generating permutations containing a bivincular pattern

Bivincular patterns have three restrictions: a relative order, adjacencies of positions, and consecutiveness of values. Hence, we use step C' and change step A to A' in a similar way as C was changed to C'. If the i-th and the $(i+1)$-th values must be consecutive, we define $\mathbb{I}'_{i,j} = \mathbb{I}'_{i+1,j+1}$.Swap$(j+1, i+1)$.

Summary of our algorithms

Our algorithms can be summarized as follows. First, we construct $\mathbb{S}_n$. Next, we construct the πDD for $C_n(\sigma)$ by choosing the steps to take according to the pattern to avoid. Finally, we calculate the set difference of S_n and $C_n(\sigma)$ by using πDDs, and hence obtain the πDD for $Av_n(\sigma)$. This procedure is illustrated in Figure 4.15.

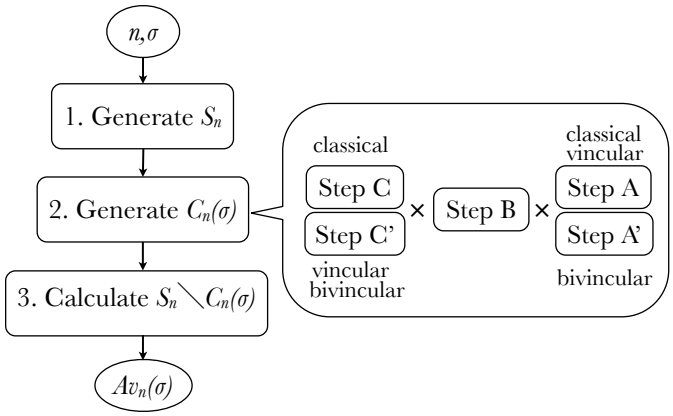


Figure 4.15. The summary of our algorithms with πDDs

4.4.4 Rot-πDD-based method

In this subsection, we provide construction algorithms of Rot-πDDs for $\mathbb{A}$ and $\mathbb{C}$. Then, we estimate and compare the size of πDDs and Rot-πDDs for $\mathbb{A}$ and $\mathbb{C}$.

Construction of Rot-πDD $\mathbb{C}$

As stated in the previous subsection, a Rot-πDD $\mathbb{C}$ contains only permutations π such that there are k indices $1 \leq p_1 < p_2 < \cdots < p_k \leq n$ with $\pi_{p_i} = i$. The composition $\rho_{1,p_1} \cdot \rho_{2,p_2} \cdot \cdots \cdot \rho_{k,p_k}$ is an instance of such π. This composition can be calculated as Definition 4.1.2: we start ι, and then left-rotate the interval $[k, p_k]$, and next left-rotate the interval $[k-1, p_{k-1}]$, and so on. Here, intervals $[k, p_k]$ are valid since $k \leq p_k$ holds. In addition, for each left-rotation ρ_{k,p_k}, it is guaranteed that k is at the k-th position since elements before the k'-th position are not moved by $\rho_{k',p_{k'}}$ $(k < k')$. These facts ensure that each left-rotation ρ_{k,p_k} moves k to the p_k-th position.

The above observation gives the following recursion: Let $Pos_{i,j}$ be a set of permutations such that the largest left-rotation of their left-rotation decompositions is $\rho_{j,i}$ or smaller, then we can add $\rho_{j+1,i+1}$ to the end of each permutation in $Pos_{i,j}$, and obtain $Pos_{i+1,j+1}$. $Pos_{n,k}$ is the desired permutation set. Algorithm 4.4.3 realizes this recursion by dynamic programming. Note that this algorithm quite similar to Algorithm 4.4.1: only one different point is using LeftRot instead of Swap at line 6.

Construction of Rot-πDD $\mathbb{A}$

Let $\mathbb{A}$ be the Rot-πDD for the set of all n-permutations whose k-prefix is ordered in the increasing order. Here, we can observe that the left-rotation decomposition of such permutations does not include $\rho_{i,j}$ $(j \leq k)$ since if such $\rho_{i,j}$ exists, the new j-th

Algorithm 4.4.3 Construct Rot-πDD $\mathbb{C}$.

```
1: procedure CONSTRUCTC(n,k)
2:     P_{i,j} ← the 1-sink for j = 0, otherwise the 0-sink
3:     for j = 1 to k do
4:         for i = j to n do
5:             P_{i,j} ← P_{i-1,j} ∪ P_{i-1,j-1}.LeftRot(j,i)
6:         end for
7:     end for
8:     return P_{n,k}
9: end procedure
```

Algorithm 4.4.4 Construct Rot-πDD $\mathbb{A}$.

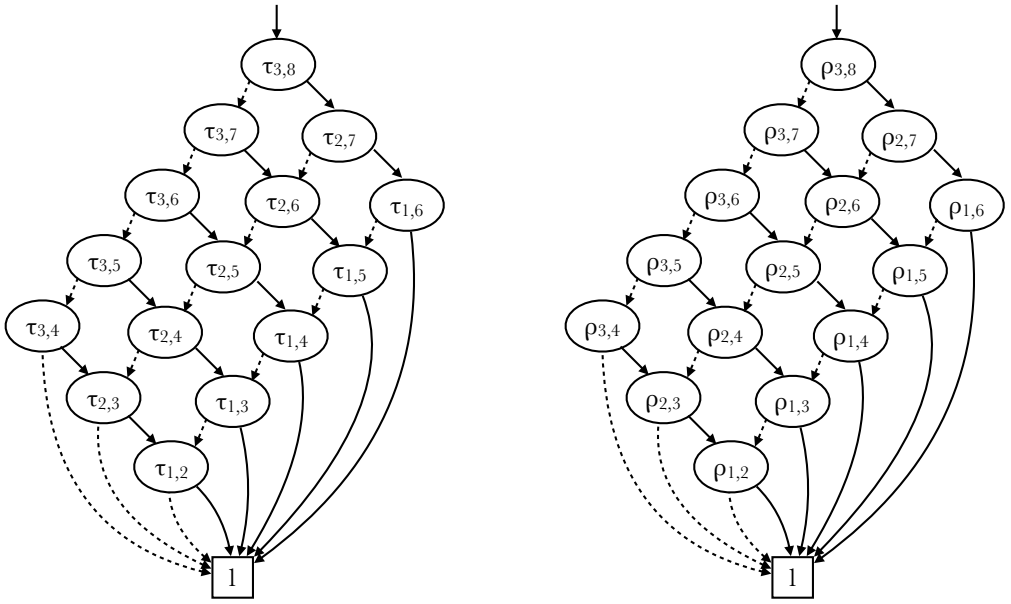
```
1: procedure CONSTRUCTA(n,k)
2:     I_k ← the 1-sink
3:     for i = k+1 to n do
4:         I_i ← I_{i-1}
5:         for j = 1 to i-1 do
6:             I_i ← I_i ∪ I_{i-1}.LeftRot(j,i)
7:         end for
8:     end for
9:     return I_n
10: end procedure
```

element must be smaller than the previous j-th element, and thus the k-prefix is not in the increasing order. On the other hand, $\rho_{i,l}$ $(k < l)$ does not disturb the k-prefix increasing sequence as shown in Section 4.2.3. Therefore, all the permutations whose left-rotation decomposition consists only of $\rho_{i,l}$ is in $\mathbb{A}$. Consequently, a permutation π is in $\mathbb{A}$ if and only if π can be decomposed into left-rotations except $\rho_{i,j}$ $(j \leq k)$. Algorithm 4.4.4 describes construction of the Rot-πDD including only such permutations.

Sizes of πDDs and Rot-πDDs for $\mathbb{A}$ and $\mathbb{C}$

In Algorithm 4.4.1 (resp. Algorithm 4.4.3), the top label of $\mathbb{P}_{i,j}$ is $\tau_{j,i}$ (resp. $\rho_{j,i}$) for $j < i$[5]. This indicates that $\mathbb{P}_{i-1,j-1}$.Swap(j,i) (resp. LeftRot(j,i)) just places $\tau_{j,i}$ (resp. $\rho_{j,i}$) on the top of $\mathbb{P}_{i-1,j-1}$, and increases the size only by one. Thus, the size of $\mathbb{P}_{i,j}$ $(j < i)$ is $j(i-j)$ (except the 1-sink), and the size of $\mathbb{C}$ is $k(n-k) = O(n^2)$. Figure 4.16 illustrates the beautiful structure of $\mathbb{C}$ in the form of a πDD and a Rot-πDD, which helps intuitive understandings of the exact size of decision diagrams. It is

[5]We consider $i = j$ for the 1-sink.



(a) πDD (b) Rot-πDD

Figure 4.16. A πDD and a Rot-πDD for $\mathbb{C}$ with $n = 8$ and $k = 3$

also interesting that the πDD and the Rot-πDD for $\mathbb{C}$ has the same shape although they are based on different decomposition manners, because $\tau_{i,i+1}$ and $\rho_{i,i+1}$ are the same permutation: an adjacent transposition.

Size estimation of πDD $\mathbb{A}$ from Algorithm 4.4.2 seems to be difficult due to Cartesian product operation. In addition, the experimental result shown later in Section 4.4.5 indicates that the size of πDD $\mathbb{A}$ is large even if n and k are small. In contrast, Algorithm 4.4.4 is quite simple and has the same property for construction of $\mathbb{C}$: LeftRot(j,i) just places $\rho_{j,i}$ on the top of I_{i-1}. Thus the number of nodes of $\mathbb{A}$ is equal to the number of the steps in Algorithm 4.4.4, i.e., $n(n-1)/2 - k(k-1)/2 = O(n^2)$. The structure of Rot-$\pi$DD $\mathbb{A}$ is also regular as shown in Figure 4.17.

Here, all of the Rot-πDDs $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$ have polynomial size. Although the exact time complexity of Cartesian product operation has never been revealed, #P-completeness of the counting problem of pattern-avoiding permutations suggests that Cartesian product operation seems not to be a polynomial time operation.

4.4.5 Experimental Results

We implemented our algorithms in C++ and carried out computational experiments on a 3.20 GHz Intel Core i7-3930K CPU machine with Ubuntu 14.04 LTS 64-bit OS and 64 GB memory.

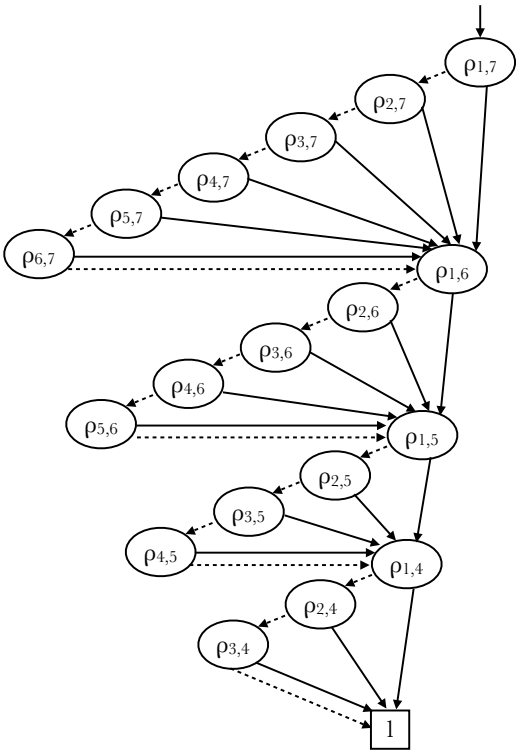


Figure 4.17. A Rot-πDD for $\mathbb{A}$ with $n = 7$ and $k = 3$

Comparison of the Sizes of Decision Diagrams for $\mathbb{A}$

We compare the sizes of πDDs and Rot-πDDs for $\mathbb{A}$ by experiments. Table 4.4 summaries the sizes of each decision diagram representing $\mathbb{A}$ for $n = 11$ to 15. The sizes of Rot-πDDs are exactly $n(n-1)/2 - k(k-1)/2$ as estimated in the previous subsection. On the other hand, the sizes of πDDs are extremely larger than the sizes of Rot-πDDs for all pairs n and k; in the worst case, the difference of the sizes between a πDD and a Rot-πDD is four orders of magnitude.

Comparison of the Performance of Enumeration

We conducted experiment to measure the practical performance of our algorithms. We also compared the performance of our algorithms to the following methods:

1. Naïve method: generates all n-permutations and, for each n-permutation, decides whether it contains σ or not by checking the order isomorphism between all k-subsequences and σ.

2. PermLab method: repeatedly extend σ-avoiding permutations with length $i-1$ to σ-avoiding permutations with length i, 1 through n. Each extension is accelerated by pruning the position at which insertion of i will be failed.

Table 4.4. The size of πDDs and Rot-πDDs for $\mathbb{A}$ (except the 0-/1-sinks)

	n									
	11		12		13		14		15	
k	πDD	Rot-πDD	πDD	Rot-πDD	πDD	Rot-πDD	πDD	Rot-πDD	πDD	Rot-πDD
2	11910	54	35220	65	111230	77	347688	90	1151934	104
3	11904	52	35214	63	111224	75	347682	88	1151928	102
4	11856	49	35166	60	111176	72	347634	85	1151880	99
5	11496	45	34806	56	110816	68	347274	81	1151520	95
6	8616	40	31926	51	107936	63	344394	76	1148640	90
7	4116	34	19326	45	82736	57	319194	70	1123440	84
8	1312	27	7506	38	40256	50	198234	63	881520	77
9	303	19	2014	30	12938	42	78474	55	440520	69
10	55	10	405	21	2974	33	21252	46	144360	60
11	0	0	66	11	528	23	4250	36	33498	50
12	—	—	0	0	78	12	674	25	5906	39
13	—	—	—	—	0	0	91	13	845	27
14	—	—	—	—	—	—	0	0	105	14
15	—	—	—	—	—	—	—	—	0	0

Tables 4.5 and 4.6 show the results for generating permutations avoiding a classical pattern. The tables show the best, the worst, and the average computation time and memory consumption over all patterns with length $k = 3$, 4, and 5. Note that computation time of our πDD and Rot-πDD methods is time to construct the decision diagram for $Av_n(\sigma)$, and computation time of the naive and PermLab methods is time to output all pattern-avoiding permutations to /dev/null.

For computation time, the naïve method cannot finish computation even if the length of patterns is small. The runtime grows ten-folds or more with respect to the length of texts. PermLab works averagely three-folds better than πDD method for patterns with length 3. However, the performance for longer patterns, i.e. the length greater than 3, is worse than the performance of the πDD method. For example, in the case of $n = 14$ and $k = 4$, the πDD method averagely requires only 10% of the time required by the PermLab method. Using Rot-πDDs dramatically improves the performance of the πDD method: the Rot-πDD method is two orders of magnitude faster than the πDD method in the case $n = 15$ and $k = 4$, and computation time of the Rot-πDD method for $n = 20$ and $k = 4$ cases is less than computation time of the πDD method for $n = 15$ and $k = 4$.

It should be noted that there are differences between the best and worst performance for pattern with the same length in the results of decision diagram-based algorithms, while the naïve and PermLab methods hardly shows any differences. However, in almost all worst-case scenarios, the performance of the Rot-πDD method is better than the best-case scenario of the search-based ones.

Memory consumption of the πDD and Rot-πDD method is significantly small compared to the number of permutations in the decision diagrams. This indicates that compression of πDDs and Rot-πDDs is effective for pattern-avoiding permutations. Rot-πDDs show higher compression ratio than πDD other than the best cases of $k = 3$. The results intend that runtime and memory consumption seem to have the proportional relation.

Table 4.5. Computation time (second) for generating classical pattern-avoiding permutations

		Naïve Method			PermLab Method			πDD Method			Rot-πDD Method		
		k			k			k			k		
n		3	4	5	3	4	5	3	4	5	3	4	5
	best	1.140	6.000	17.089	0.002	0.028	0.091	0.004	0.016	0.024	0.004	0.005	0.004
10	average	1.173	6.453	18.223	0.003	0.034	0.112	0.011	0.028	0.036	0.005	0.007	0.009
	worst	1.200	6.932	19.697	0.006	0.056	0.194	0.020	0.044	0.060	0.006	0.010	0.017
	best	13.053	89.494	417.958	0.005	0.173	0.825	0.012	0.044	0.092	0.009	0.011	0.010
11	average	13.216	95.727	435.211	0.007	0.201	0.985	0.029	0.101	0.174	0.009	0.018	0.018
	worst	13.365	101.878	453.604	0.011	0.298	1.220	0.052	0.152	0.276	0.010	0.026	0.026
	best	171.954	—	—	0.016	1.199	8.053	0.024	0.152	0.428	0.020	0.022	0.024
12	average	174.018	—	—	0.018	1.358	9.806	0.087	0.444	0.921	0.022	0.031	0.040
	worst	175.934	—	—	0.020	1.611	11.697	0.156	0.780	1.392	0.028	0.039	0.063
	best	—	—	—	0.061	8.324	78.190	0.048	0.568	1.824	0.044	0.053	0.052
13	average	—	—	—	0.068	9.277	98.256	0.284	1.787	4.309	0.050	0.066	0.100
	worst	—	—	—	0.077	10.801	130.391	0.544	3.072	6.888	0.054	0.090	0.186
	best	—	—	—	0.217	54.934	—	0.116	1.960	6.448	0.115	0.118	0.115
14	average	—	—	—	0.233	65.989	—	1.029	6.769	19.036	0.123	0.158	0.305
	worst	—	—	—	0.263	79.993	—	1.968	12.021	32.370	0.140	0.242	0.611
	best	—	—	—	0.775	462.942	—	0.300	5.688	23.814	0.301	0.306	0.310
15	average	—	—	—	0.852	381.689	—	3.513	24.771	85.655	0.320	0.422	0.912
	worst	—	—	—	0.943	585.620	—	6.860	48.415	160.562	0.354	0.610	1.990
	best	—	—	—	2.765	—	—	0.705	15.110	70.437	0.840	0.822	0.838
16	average	—	—	—	3.002	—	—	10.795	78.113	300.129	0.862	1.110	2.673
	worst	—	—	—	3.412	—	—	21.002	157.880	585.190	0.887	1.575	6.530
	best	—	—	—	10.945	—	—	1.807	43.357	—	2.130	2.093	2.095
17	average	—	—	—	12.282	—	—	37.014	283.071	—	2.179	2.748	7.557
	worst	—	—	—	13.484	—	—	73.622	598.761	—	2.250	3.817	19.078
	best	—	—	—	41.322	—	—	4.231	—	—	5.230	5.075	5.031
18	average	—	—	—	43.164	—	—	127.887	—	—	5.291	6.552	20.604
	worst	—	—	—	47.421	—	—	254.791	—	—	5.407	9.116	60.025
	best	—	—	—	147.063	—	—	10.283	—	—	12.528	12.008	11.910
19	average	—	—	—	155.777	—	—	445.950	—	—	12.727	15.307	56.159
	worst	—	—	—	171.360	—	—	906.655	—	—	13.175	21.177	198.094
	best	—	—	—	561.707	—	—	—	—	—	29.717	28.713	28.074
20	average	—	—	—	595.214	—	—	—	—	—	29.874	34.986	154.494
	worst	—	—	—	654.233	—	—	—	—	—	30.169	47.085	680.858

Table 4.6. Memory consumption (kB) for generating classical pattern-avoiding permutations.

		$\#Av_n(\sigma)$			πDD Method			Rot-πDD Method		
		k			k			k		
n		3	4	5	3	4	5	3	4	5
	best				2760	4212	7156	2360	2360	2360
10	average	16796	574150	2171460	3509	6856	7376	2361	2706	2862
	worst				4260	7444	7700	2364	2956	2956
	best				4208	7676	13720	2952	2960	2956
11	average	58786	3648275	19011623	5876	17641	25233	2954	4139	4354
	worst				6792	25084	27000	2956	4624	4644
	best				7112	25316	49108	4640	4640	4640
12	average	208012	23771768	173553425	16106	48405	93525	5075	7162	9432
	worst				25084	94788	102940	7244	7792	14948
	best				12708	51436	188416	7836	13112	13172
13	average	742900	158260498	1641499314	32130	168848	337123	9850	14078	19753
	worst				51084	201264	408460	14068	14996	26664
	best				24440	187372	396432	24848	24972	24984
14	average	2674440	1073474327	16006197603	111634	542621	1282547	25659	27141	43704
	worst				190460	779640	1587600	26156	48472	96904
	best				47620	389140	1543108	48748	48404	48860
15	average	9694845	7401901167	160274747099	234836	1560720	4986352	49296	59769	107354
	worst				406440	3092572	6471388	49756	97140	197228
	best				94488	807736	5924216	97504	97224	97280
16	average	35357670	51789495305	1642837274942	863992	5477517	19534358	97789	129188	251457
	worst				1571096	12058376	25328368	98324	193140	758348
	best				189892	3016896	—	195956	196272	196136
17	average	129644790	367152104849	13729671069165	3226830	18970784	—	196482	274738	656590
	worst				6134872	47290404	—	197384	386564	1558264
	best				378448	—	—	395404	390964	390652
18	average	477638700	2634072644232	104591644374404	12331948	—	—	396304	553176	1564507
	worst				23845996	—	—	398492	778860	3307180
	best				770112	—	—	814724	802256	799584
19	average	1767263190	15210871435804	670098267999396	26406442	—	—	925279	1263493	4022444
	worst				51401392	—	—	1477460	1598624	12509448
	best				—	—	—	2984852	2979536	2958408
20	average	6564120420	72990438715891	3747149016070295	—	—	—	2985622	3055231	10593213
	worst				—	—	—	2986956	3188512	48122832

4.4.6 Concluding Remarks for Pattern-Avoiding Permutations

In this section, we proposed an algorithm for generating pattern-avoiding permutations using πDDs and Rot-πDDs. Our proposed method is easily extended to enable to process generalized patterns such as vincular pattern and bivincular patterns. In addition we showed that the exact sizes of Rot-πDDs in the middle of the algorithm are $O(n^2)$. Experimental results demonstrate that proposed algorithms are faster than the search-based method and costs less memory than the naive storing. Especially, the Rot-πDD-based method significantly well-performed: the Rot-πDD-based method runs two or more orders of magnitude faster than the algorithm used in PermLab, a major software for manipulation of pattern-avoiding permutations.

Future work is to improve further the computation time and memory consumption of our algorithm, and to compare our algorithm and other algorithms for some particular patterns, for example Baxter permutations [7]. Moreover, we are also interested in analyzing the relationship between pattern-avoiding permutations and floorplans. In future work, we plan to develop several functions such as search by criteria and random sampling to use Rot-πDDs as floorplan databases.

4.5 Proper Usage of Permutation Decision Diagrams

We have seen several applications of permutation decision diagrams, and they include both of the applications suitable to πDD and the applications suitable to Rot-πDDs. It is natural that we want to reveal what properties of problems distinguish πDD-suitable problems and Rot-πDD-suitable problems, and how we choose πDDs and Rot-πDDs as a data structure for a given problem.

From applications in previous chapters, πDDs are preferable to problems related to swaps (reversible circuits) or cycles (cycle-type partitions). On the other hand, Rot-πDDs are preferable to problems solved by a dynamic programming approach on subsets (Eulerian trails and topological orders) or related to relative orders (pattern-avoiding permutations). Although these can be hints to choose decision diagrams, it is more useful if we know key factors in terms of mathematical characterizations of problems.

In this section, we tackle a proper selection problem of permutation decision diagrams. We first review studies on compression of a single permutation to obtain hints to reveal the relation between compression ratio and permutation parameters. Direct extension of such study to the decision diagram selection problem seems to be difficult because we compress a permutation set, not a single permutation. We conduct preliminary experiments to empirically analyze what parameters of permutations in a set affect the compression ratio of πDDs and Rot-πDDs.

4.5.1 Compression of a Permutation

In general, naïve representation of a permutation, i.e. an array of integers, is asymptotically optimal: since the number of all n-permutations is $n!$, the lower bound of the length of bits to represent an n-permutation is $\log(n!) \simeq n\log n - 1.44n$, which can be asymptotically achieved by an array with n integers each of which represented by $\lceil \log n \rceil$ bits. Does this mean that we have nothing to do anymore?

Barbay and Navarro [5] state the relation between permutation compression and *adaptive sort* [29]. A sorting algorithm is adaptive for a *disorder measure* if the algorithm runs fast for permutations with small disorder measure, where disorder measures characterize sorting-hardness of permutations in some sense. For example, the number of *inversions*, pairs (i, j) of indices $i, j (i < j)$ satisfying $\pi_i > \pi_j$ is a disorder measure of permutations. More formally, we define optimally adoptive sorting as follows:

Definition 4.5.1. *Let M be a function for a permutation such that $M(\pi)$ is a disorder measure of a permutation π. Then a sorting algorithm is* optimally adaptive *with respect to M if for any n-permutation π, the number of comparison in the algorithm is at most $O(\max\{n, \log|B(M(\pi), n, M)|\})$, where $B(k, n, M)$ is the set of permutations σ with length n or less such that $M(\sigma) \leq k$.*

We notice that the theoretical lower bound $\Omega(n \log n)$ of sorting algorithms comes from the fact that sorting algorithms need $\lceil \log(n!) \rceil$ comparisons to distinguish $n!$ permutations. Conversely, optimally adaptive sorting algorithms can distinguish a permutation with a small disorder measure in the small number of comparisons, and thus yields a short comparison sequence. Since we can decode a permutation from (short) comparison sequences, we can assign comparison sequences as short compressed codes to permutations with small disorder measures. Barbay and Navarro [5] have indeed proposed encoding scheme adaptive to a disorder measure *runs*, one of disorder measures, with indexing to calculate $\pi()$ and $\pi^{-1}()$.

Here, we introduce the six disorder measures:

Definition 4.5.2. *Inv:* Inversions *of a permutation π are pairs of two indices (i, j) $(1 \leq i < j \leq n)$ such that $\pi_i > \pi_j$. We denote the number of inversions of π as $Inv(\pi) = |\{(i, j) \mid 1 \leq i < j \leq n, \pi_i > \pi_j\}|$.*

Example 4.5.1. *Let $\pi = (4,2,3,1,6,5)$, then $Inv(\pi) = 6$ because there are inversions $(1,2),(1,3),(1,4),(2,4),(3,4),$ and $(5,6)$ in π.*

Definition 4.5.3. *Dis: $Dis(\pi)$ is defined as the maximum distance between two elements of an inversion in π. Formally, $Dis(\pi) = \max\{j - i \mid 1 \leq i < j \leq n, \pi_i > \pi_j\}$.*

Example 4.5.2. *Let $\pi = (4,2,3,1,6,5)$, then $Dis(\pi) = 3$ because the distance of the inversion $(1,4)$ in π is 3 and this is the maximum distance.*

Definition 4.5.4. *Max: $Max(\pi)$ is defined as the maximum distance between the position of an element in π and its position in ι. Formally, $Max(\pi) = \max\{|\pi_i - i| \mid 1 \leq i \leq n\}$.*

Example 4.5.3. *Let $\pi = (4,2,3,1,6,5)$, then $Max(\pi) = 3$ because $\pi_1 = 4$ gives the maximum distance.*

Definition 4.5.5. *Exc: $Exc(\pi)$ is defined as the minimum number of exchanges to sort π. In other words, $Exc(\pi)$ is the minimum number of transpositions to make π by the composition of the transpositions. It is known that the number of required exchanges to sort an n-permutation π equals $n - \sum_{i=1}^{n} \vec{c}_i^{\pi}$, i.e., n minus the number of cycles in π. Thus, $Exc(\pi) = n - \sum_{i=1}^{n} \vec{c}_i^{\pi}$ for an n-permutation π.*

Example 4.5.4. *Let $\pi = (4,2,3,1,6,5)$, then $Exc(\pi) = 2$ because $\pi = (1\ 4)(2)(3)(5\ 6)$ consists of four cycles.*

Definition 4.5.6. *Rem: $Rem(\pi)$ is defined as the minimum number of removed elements in π to make π an ascending sequence. It is known that the minimum number of removed elements of an n-permutation equals $n - Lis(\pi)$, where $Lis(\pi) = \max\{k \mid 1 \leq i_1 < \cdots < i_k \leq n, \pi_{i_1} < \cdots < \pi_{i_k}\}$ is the length of the longest increasing subsequence in π. Thus, $Rem(\pi) = n - Lis(\pi)$ for an n-permutation π.*

Example 4.5.5. *Let* $\pi = (4,2,3,1,6,5)$*, then* $Rem(\pi) = 3$ *because* π *has the longest increasing subsequence with length three, e.g.* $(2,3,6)$.

Definition 4.5.7. *Runs: Runs in an n-permutation* π *is a set of* step-downs *in* π*, where an index i is a step-down if* $\pi_i > \pi_{i+1}$*. We denote the number of runs of* π *as* $Runs(\pi) = |\{i \mid 1 \leq i < n, \pi_i > \pi_{i+1}\}|$

Example 4.5.6. *Let* $\pi = (4,2,3,1,6,5)$*, then* $Runs(\pi) = 3$ *because* $\{1,3,5\}$ *is the runs in* π.

We can find them in the survey paper of adaptive sorting [29]. Note that all the disorder measures introduced in this section except Inv are in the range from 0 to $n-1$, whereas Inv is in the range 0 to $n(n-1)/2$, for n-permutations.

4.5.2 Preliminary Experiments for Disorder Measures

In this thesis, we discuss compression of a permutation set, not a single permutation. The lower bound of the size of compression for a permutation set is $\log 2^{n!} = n!$ bits. This means that we have room for reduction by a factor of $O(n \log n)$ of the size $O(n! n \log n)$ of naïve array representation. Unfortunately, theoretical analysis of the size of decision diagrams base on disorder measures seems not to be obvious. However, on intuitive grounds the author guesses that a set of permutations with small disorder measures intends to be well compressed because, for instance, permutations with small Exc are decomposed into a few transpositions, and permutations with a few left-rotations will be less disordered due to the relative-order keeping property of left-rotations.

We conducted preliminary experiments to empirically verify this prediction and shows the results in Table 4.7. The results show that decision diagrams can highly compress permutation sets not only with small measures but also with large measures, especially when we use preferable one. The author guesses that (some of) these measures have regular structure in the form of πDDs or Rot-πDDs, like for combinations in Section 4.4.3. For almost all disorder measures, Rot-πDDs are preferable to πDDs. In particular, for Inv, Rem, and Runs, Rot-πDDs succeed to compress permutation sets two orders of magnitude highly than πDDs. On the other hands, for only Exc, the compression of πDDs is 50-folds more efficient than Rot-πDDs in the best case. An interesting fact is the relation between Exc and Rem. It is known that Rem-optimally adaptive sorting is always Exc-optimal; this fact is associated that the compression ratios for Rem and Exc tend to be close each other. However preferable decision diagrams for Exc and Rem are different: πDDs are preferable for Exc, while Rot-πDDs are preferable for Rem according to the experimental results.

4.5.3 Concluding Remarks of Proper DD Selection

In this section, we consider which of πDDs and Rot-πDDs are used for each problems, focusing on characterizations based permutation parameters, especially disorder measures. Preliminary experiments indicate that regularly-disordered permutation sets are well-compressed by our decision diagrams πDDs and Rot-πDDs. In particular, Rot-πDDs are preferable to πDDs for almost all disorder measures in the experiments except Exc. This result suggests that if objective permutation sets looks like regularly disordered, we try first usage of Rot-πDDs in general. On the other hand, if a given problem seems to be related to cycles, πDDs may outperform Rot-πDDs.

In future directions, we should discuss theoretical analysis for the relation between the size of decision diagrams and permutation parameters. From results for Exc and Rem, the hierarchy of disorder measures in terms of adaptive sorting does not hold for the size of decision diagrams. On the other hand, disorder measures may be a key to analyze the compression performance of decision diagrams because decision diagrams achieve the high compression for regularly disordered permutations. The author considers that theoretical analysis for permutations with disordered measures, including measures other than ones in this thesis, is a first step to give more theoretical bounds of the size of permutation decision diagrams.

Table 4.7. Experimental results for several measures: Inv, Dis, Max, Exc, Rem, and Runs. "#perms" means the number of the permutations π that the corresponding measure of π is at most k. "πDD Size" and "Rot-πDD Size" mean the size of the πDD and the Rot-πDD representing the set, respectively. Note that 0-/1-sinks are not included in the size here.

(a) Inv

k	#perms	πDD Size	Rot-πDD Size
0	1	0	0
1	10	9	9
2	54	31	24
3	209	64	45
4	649	110	69
5	1717	173	97
6	4015	264	128
7	8504	390	158
8	16599	556	189
9	30239	771	220
10	51909	1041	250
11	84592	1365	275
12	131635	1747	299
13	196524	2189	322
14	282578	2712	344
15	392588	3311	365
16	528441	3991	380
17	690778	4734	394
18	878737	5523	407
19	1089826	6345	419
20	1319957	7182	430
21	1563651	8016	440
22	1814400	8818	443
23	2065149	9561	445
24	2308843	10211	446
25	2538974	10760	446
26	2750063	11179	445
27	2938022	11440	443
28	3100359	11511	440
29	3236212	11357	429
30	3346222	10955	417
31	3432276	10305	404
32	3497165	9425	390
33	3544208	8354	375
34	3576891	7158	359
35	3598561	5905	342
36	3612201	4661	324
37	3620296	3491	297
38	3624785	2461	269
39	3627083	1618	240
40	3628151	983	210
41	3628591	549	179
42	3628746	278	147
43	3628790	126	114
44	3628799	61	80
45	3628800	45	45

(b) Dis

k	#perms	πDD Size	Rot-πDD Size
0	1	0	0
1	89	9	9
2	1285	43	38
3	8420	141	125
4	35505	464	351
5	103050	1395	771
6	287280	2933	920
7	756000	2932	618
8	1814400	1152	240
9	3628800	45	45

(c) Max

k	#perms	πDD Size	Rot-πDD Size
0	1	0	0
1	89	9	9
2	2177	51	43
3	19708	220	143
4	95401	880	293
5	329462	2737	289
6	899064	2905	209
7	1865520	1170	133
8	2943360	273	79
9	3628800	45	45

(d) Exc

k	#perms	πDD Size	Rot-πDD Size
0	1	0	0
1	46	45	80
2	916	80	420
3	10366	105	1544
4	73639	120	3769
5	342964	125	5765
6	1066644	120	6042
7	2239344	105	5594
8	3265920	80	4131
9	3628800	45	45

(e) Rem

k	#perms	πDD Size	Rot-πDD Size
0	1	0	0
1	82	87	52
2	2603	498	141
3	40884	2039	303
4	337210	5936	513
5	1438112	10740	660
6	3042210	11321	556
7	3612004	4245	266
8	3628799	61	80
9	3628800	45	45

(f) Runs

k	#perms	πDD Size	Rot-πDD Size
0	1	0	0
1	1014	666	80
2	48854	4370	223
3	504046	9298	315
4	1814400	11908	360
5	3124754	11336	360
6	3579946	7724	315
7	3627786	2313	223
8	3628799	61	80
9	3628800	45	45

Chapter 5

Variable Ordering for High Compression

5.1 Background

A graph is an important discrete structure in both theoretical and practical areas of computer science. Many graph problems require us to find "one" solution that is optimal under some evaluation function. On the other hand, there are few results with the ability to find "all" solutions and store them, because practically the number of solutions may be huge, i.e., exponential to the graph size. However, if we store all solutions, we can analyze and manipulate them to a much greater extent: counting, random sampling, extraction with criteria, and optimization.

One way to manipulate all solutions on memory is using compressed data structures. Knuth [50] have used ZDD to store all simple paths. Apart from the compactness of ZDD that may enable us to store all solutions, ZDD also has set-algebra operations that work without restoring ZDDs. Therefore, taking intersection of two ZDDs, counting the number of solutions, and finding the optimal solution are achieved in time depending only on the size of the ZDDs, rather than on the number of solutions [50].

Furthermore, Knuth [50] have proposed a ZDD construction algorithm, called *Simpath*, to directly construct a ZDD and reduce runtime for explicit simple path enumeration. *Frontier-based search* [45] is a generalized framework derived from Simpath to enumerate all subgraphs with other constraints such as no cycle, connected, and degree-bounded. Frontier-based search has been used in several applications such as grid path enumeration [42], power networks [39], and puzzle games [89].

Our goal in this chapter is to improve the efficiency of frontier-based search in order to apply this search technique to as large graphs as possible. We achieve this goal by focusing on the edge ordering of a graph because the use of frontier-based search requires us to provide an edge order to the method as input. The performance of the

method and the size of the resulting ZDD are dramatically affected by the edge order. Details are discussed in Section 5.2.

Furthermore, we indicate the relation between frontier-based search and the path width [47] of a well-known graph parameter in Section 5.2.4. This means the edge-ordering problem is hard as is the minimum path-width problem. We thus propose meta-heuristics to find an appropriate edge order for frontier-based search. Our algorithm can also be considered as a method to find path decomposition with a small path width.

We propose our method in Section 5.3. Our ordering is based on beam search with start vertices found by linear time search. The features of our method are:

- Practical: Most previous algorithms focus on only maximum path width; however, the complexity of frontier-based search is affected by the entire width. Our algorithm consciously evaluates the average width in addition.

- Scalable: Frontier-based search proceeds linear to the graph size but exponentially to the path width (as discussed in Section 5.2.3). Thus, frontier-based search is scalable for graphs that are not too large with small width, and we should obtain an edge order with small width for such graphs. Our algorithm runs in $O(|V||E|K)$, where V is a vertex set, E is an edge set, and K is a user parameter. Hence, it is expected to run for graphs with $|V|, |E| \leq 10{,}000$ in feasible time.

In Section 5.4, we provide the experimental results obtained by evaluating the practical performance of our algorithm. The results show our method succeeds in finding good path decomposition in terms of both of the maximum and average width. The results also indicate that our method improves frontier-based search.

5.2 Subgraph Enumeration and Frontier-based search

Subgraph enumeration is the enumeration of all subgraphs satisfying given constraints such as no cycle, connected, and degree bounded. For example, if a given constraint is no cycle, it requires us to enumerate all subgraphs that are forests. Since the number of subgraphs can be huge, in the worst case 2^m, listing all subgraphs may be impractical. An idea to overcome this combinatorial explosion is using compressed data structures to represent subgraphs.

ZDD facilitates enumeration of a huge number of subgraphs and their analysis; since a subgraph corresponds to a combination of edges in a graph[1], a ZDD can represent a set of subgraphs. For example, Figure 5.1 shows a ZDD representing cliques with size 3 in a complete graph with size 4. However, if we construct a ZDD by listing all

[1] if we do not matter isolated vertices.

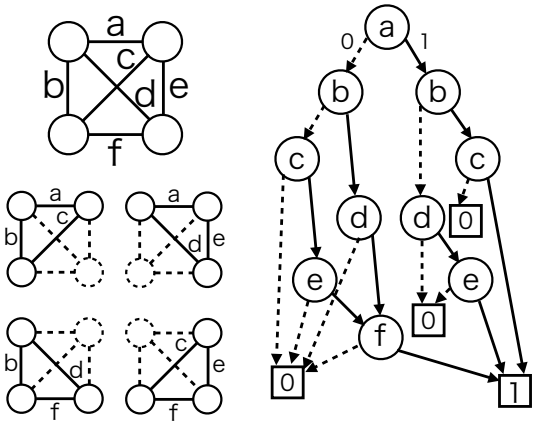


Figure 5.1. A ZDD representing all K_3 in K_4, where K_n denotes a complete graph with n vertices. Note that a ZDD has exactly one 0-sink; multiple 0's are just for visibility.

subgraphs, we indeed cannot avoid exponential calculation time. Frontier-based search is a ZDD construction framework without trying all possible solutions.

In this section, we review the algorithm of frontier method and define our problem in this chapter. Then, we proceed theoretical analysis of frontier-based search and relation between well-known graph parameter path-width, which are useful to design algorithm for our problem.

5.2.1 Algorithm Overview

Frontier-based search [45] is a ZDD construction framework. Basically, frontier-based search constructs a binary decision tree in top-down manner, deleting and sharing redundant ZDD nodes. Thus, how to delete and share the nodes is the core of the algorithm.

We identify redundant nodes by storing the "state" for each ZDD node. Stored information in states depends on given constraints for subgraphs. For example, suppose we enumerate forests. Then the information in states is the connections between pairs of vertices. If we add $e = \{u, v\}$ to a current state (i.e., use a 1-edge in a ZDD) and u and v are already connected in the current state, this is invalid because it produces cycle(s). Therefore, the 1-edge from the node with the state should point to the 0-sink. Moreover, if two ZDD nodes have the same state (e.g., the same connections in the forest case), the valid edge selections are also the same. Hence, we can share the nodes with the same state.

In the state, we have to consider only a subset of V, called a *frontier*, not all vertices in V. A frontier is a set of vertices adjacent to both of the processed edges and the unprocessed edges. More formally, for an edge order $e_1, \ldots, e_m$, the frontier F_i of the

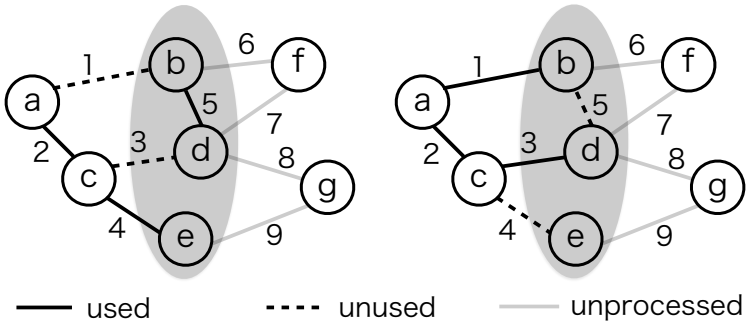


Figure 5.2. Equivalent states in the middle of frontier-based search to enumerate all sub-forests.

i-th level is defined by $F_i = adj(\{e_1, \ldots, e_i\}) \cap adj(\{e_{i+1}, \ldots, e_m\})$, where $adj(E) = \bigcup_{e \in E} e$ is the set of vertices adjacent to at least one edge of E. Figure 5.2 illustrates an example of frontiers of the 5-th level. Again, suppose we enumerate forests. Then we can consider the two subgraphs in Figure 5.2 to have the same state, because b and d are connected but e is not connected with b and d in both of the subgraphs.

Note that a ZDD produced by frontier-based search may not be well-reduced. We should use a reduction algorithm (e.g., Algorithm R in [50]) after frontier-based search, which runs in linear time in the size of a non-reduced ZDD.

For other constraints such as cliques and connected spanning graphs, we would need to use state and pruning rules other than those discussed in this paper. [45] provides a list of problems that can be solved by frontier-based search with examples of the state and the pruning rules.

5.2.2 Our Problem

Our goal is to improve the performance of frontier-based search. There are several ways affecting the performance of frontier-based search: changing state definition, adding sophisticated pruning rules, and using an appropriate edge order.

We focus on edge ordering to accelerate frontier-based search because:

- The other ways cannot be considered separately from the given constraints. On the other hand, we can determine an edge order only from a given graph, although the best edge order may also depend on the given constraints.
- A fully reduced ZDD has a canonical form unless it changes its edge order. That is, the other approaches cannot be used to change the size of the reduced ZDD, whereas edge ordering can decrease the size.

Hence, our goal is to determine a good edge order only from a given graph such that the order makes frontier-based search faster and the resulting ZDD smaller.

5.2.3 Theoretical Analysis

To improve the performance of frontier-based search, we should deeply understand what factors essentially affect the performance. Time complexity of frontier-based search is bounded by the number of states and the size of states. For example, a state for enumeration of sub-forests involves connected components between the vertices in each frontier F. Thus, the number of states in the level i is bounded by the number of set partitions of a frontier F_i, which is known as Bell number $B_{|F_i|}$. The size of states, i.e., the size of information required to distinguish it from other states, is also bounded by $O(\log B_{|F_i|})$. Hence, the time complexity of frontier-based search to enumerate all sub-forests is bounded by $O(\sum_{i=1}^{m} B_{|F_i|} \log B_{|F_i|})$. On the other hand, the size of a ZDD is $O(\sum_{i=1}^{m} B_{|F_i|})$, because a ZDD does not have to store state information after construction.

The complexity of frontier-based search is generally bounded by $O(\sum_{i=1}^{m} f(|F_i|))$, where $f(x)$ is a function determined by state definition, and exponential in x in many problems. Thus, we guess edge orders yielding small frontiers are good.

5.2.4 Frontier-based search and Path Decomposition

The definition of a frontier for edge ordering is almost the same as a *vertex separator* for vertex ordering. The j-th vertex separator S_j on the vertex order $v_1, \ldots, v_n$ is defined as $S_j = \{v_i \mid i \leq j, \exists k > j, \{v_i, v_k\} \in E\}$. We assume the edge order $e_1, \ldots, e_m$ such that for $e_x = \{v_i, v_j\}$ $(i < j)$ and $e_y = \{v_k, v_l\}$ $(k < l)$, $x < y$ if $j < l$. Then frontier F_x with $e_x = \{v_i, v_j\}$ $(i < j)$ satisfies $F_x \subseteq S_{j-1} \cup \{v_j\}$. It is because $\bigcup_{1 \leq i \leq x} e_i = \{v_1, \ldots, v_j\}$ holds and v_k $(k < j)$ is in F_x if v_k has at least one unprocessed edge, which is $\{v_k, v_l\}$ $(k < j \leq l)$, and thus v_k satisfies the definition of vertex separator S_{j-1}. Furthermore, e_x adds only v_j to S_{j-1}, and not v_i because v_i is already in S_{j-1} due to $i < j$. Therefore, $|F_x| \leq |S_{j-1}| + 1$ holds when we use the above edge order. The relation between vertex separators and a frontier-based search on BDD has been discussed in [74], for example.

Furthermore, it is known that the maximum size of vertex separators is equivalent to the path width of a corresponding *path decomposition* [47]. Path decomposition of $G = (V, E)$ is a sequence $(X_1, \ldots X_l)$ of subsets of V, called bags, satisfying the following requirements:

- For each edge $e = \{u, v\} \in E$, there is at least one bag X_i such that $u, v \in X_i$.
- For each vertex $v \in V$, if there are two bags $X_i, X_j (i < j)$ both of which contain v, for all $i \leq k \leq j$, X_k also contains v.

Path-width of a path decomposition is the maximum size $\max_{1 \leq i \leq l} |X_i| - 1$ of bags. Therefore, path decomposition with a small path width seems to yield a good vertex order.

In the context, frontier-based search can be considered as a dynamic programming (DP) algorithm on a path decomposition. But we purposely use frontier-based search rather than DP on a path decomposition because:

- We can consider a non-reduced ZDD made by frontier-based search is equivalent to a DP table of a path decomposition. Thus, the reduced ZDD has the smaller size than a DP table, which will accelerate DP calculation (i.e. traversal of the DAG). This is a merit especially when we repeatedly compute different objective functions and edge-weights dynamically change.
- If there are multiple constraints, designing DP on a path decomposition may be complex. ZDD framework makes it easy by constructing ZDDs for each constraint and taking intersections. Moreover, we can also use *subsetting* technique [41], which can compute intersections may be faster than ordinary intersection operation.

5.3 Proposed Method

A path decomposition with small path width will yield a good vertex order for a good edge order to frontier-based search. We review previous work related to the computation of a small path width before proceeding with our method.

5.3.1 Previous Work for Path Decomposition

Lengauer [52] showed that computing the minimum of the maximum vertex separator is NP-complete, and Kinnersley [47] showed this problem is equivalent to finding the minimum path width. Thus, two goals have been mainly discussed in the literature: (1) computing the exact optimal width as fast as possible and (2) computing as good a solution as possible within feasible time by heuristics.

For an exact solution, there are several results providing polynomial time algorithms for restricted graph classes [10], [35], [66]. For general graphs, Coudert et al. [23] proposed an algorithm based on branch and bound. The paper [23] showed this algorithm to be faster than the SAT-based algorithm in most cases.

As heuristics, several linear time algorithms have been proposed such as:

- DFS/BFS [61]: Vertices are ordered by a depth-/breadth-first traversal.
- NDS [61]: For the order $v_1, \ldots, v_{i-1}$, let $S = \{v_1, \ldots, v_{i-1}\}$. Then a vertex $v \in V \setminus S$ with maximum $|N(v) \cap S|$ is chosen as the next vertex v_i.

- DLU[2] [27]: For the order $v_1, \ldots, v_{i-1}$, let $S = \{v_1, \ldots, v_{i-1}\}$. Then a vertex $v \in V \setminus S$ with maximum $|N(v) \cap S| - |N(v) \cap (V \setminus S)|$ is chosen as v_i.

These heuristics must choose a start vertex (typically, with the smallest degree). Many researchers using frontier-based search use BFS ordering as input. For instance, the Graphillion package [38], which is a library using frontier-based search, supports DFS/BFS ordering and sets BFS ordering as default.

On the other hand, Duarte et al. [27] proposed a meta-heuristic algorithm based on *basic variable neighborhood search* (BVNS). BVNS repeats local search with random shakes of the current best within the time limit. BVNS uses the evaluation function such that the smaller number of large bags is highly evaluated; Thus, BVNS reduces not only the maximum size but also the entire size.

5.3.2 Overview of Proposed Method

We focus on improving the practical performance of frontier-based search. Namely,

- Our guess is that the graph size that can be accommodated by frontier-based search is $n, m \leq 10{,}000$ and a path width up to 20, based on the complexity. Thus, algorithms should require polynomial time. On the other hand, if it runs in linear time, we can extend the computation to improve the quality.
- Typical optimization focuses on minimizing the maximum width. However, the complexity of frontier-based search is $O(\sum_{i=1}^{m} f(|F_i|))$. Thus, we should also be concerned about the entire size, not only the maximum one.

Our algorithm consists of three parts:

1. Calculate the appropriate start vertices by using linear time heuristics.
2. Calculate a good vertex order by using beam search with start vertices.
3. Calculate a good edge order from the calculated good vertex order.

We first introduce the core of our algorithm, beam search. Since the preliminary experiment shows the performance of beam search depends on a start vertex, we also propose a strategy to choose appropriate start vertices. Finally, we propose an algorithm to obtain a good edge order from a vertex order.

[2]Although this algorithm is called C1 in [27], we call it DLU (standing for the difference between labeled neighbors and unlabeled neighbors).

5.3.3 Core Algorithm: Beam Search

We use beam search to compute a good vertex order. Beam search is an algorithm for pruning search space with an evaluation function. The beam search traverses the search space in a breadth-first manner and expands only the top-K evaluated states in the same search level. Here, K is a user parameter called *beam width*.

The i-th step of our beam search determines the i-th vertex as follows: We already have K orders consisting of $i-1$ vertices. For each order, we try to add a vertex $v \in N(F)$, where F is the current frontier. Then we obtain a new list of orders consisting of i vertices. We extract the top-K orders with an evaluation function, and proceed to the next $i+1$-th step. Our method finally outputs the most highly evaluated order at the n-th step.

We use the sum of the squares of frontier sizes $\sum_{k=1}^{i} |F_k|^2$ as evaluation function for the i-th step, where smaller values are highly evaluated. This is for decreasing the maximum frontier size and average frontier size simultaneously. For tiebreak, we use another function $|N(F_i) \cap (\{v_{i+1}, \ldots, v_n\})|$, smaller is highly evaluated. This is because (1) neighbors may become a future frontier and (2) decreasing candidates helps to reduce expansions, i.e., it saves runtime.

The time complexity is $O(nK(n+m))$: The algorithm proceeds n steps. In each step, it examines K search nodes. Each expansion of a search node yields at most n search nodes, and updates evaluation in $O(|N(v)|)$ time for added vertex v. Thus, each expansion costs $O(n+m)$ time in total. There are at most nK new search nodes, and the top-K extraction is achieved in linear time $O(nK)$.

5.3.4 Improvement by Using New Light Search Algorithm: RFS

Preliminary experiments indicate the performance of beam search deeply depends on the start vertex. Hence, we want to choose likely appropriate vertices and attempt using them as start vertices. In order to obtain appropriate vertices, we propose to use linear time heuristics. We can run the linear time algorithm n times for our target graph size. Thus, we evaluate all vertices in terms of the maximum and average frontier size of the resulting order of heuristic search using it as a start vertex. The top-L vertices, where L is a user parameter, are chosen as appropriate vertices for beam search; then we run beam search L times with each start vertex, respectively, and output the best one among the L results.

Another advantage of using heuristics is that they enable us to output the best result of the heuristics as our solution, if it is more accurate than the best result of the beam search. Hence, more sophisticated linear time heuristics is preferable to improve the result.

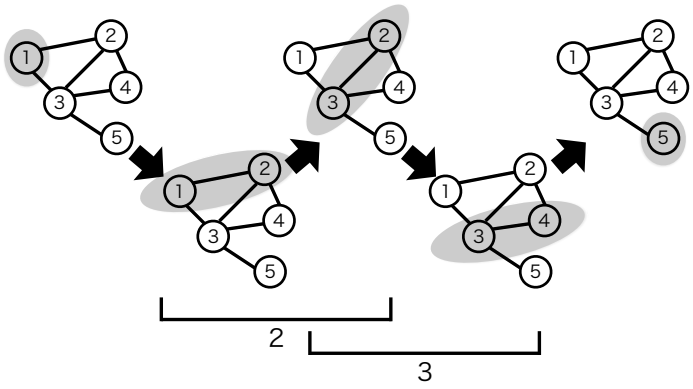


Figure 5.3. Frontier transitions and vertex intervals

We propose a new heuristics *Frontier-Reducing Search* (*FRS*): For the order $v_1, \ldots, v_{i-1}$, let $S = \{v_1, \ldots, v_{i-1}\}$. First, we choose $v \in S$ with minimum $|N(v) \cap S|$.[3] Then a vertex $u \in N(v) \cap (V \setminus S)$ with minimum $|N(u) \cap (V \setminus S)|$ is chosen as the next vertex. This intends to choose a vertex that appears to be easily removed from the current frontier to keep frontier size small. Experiments in Section 5.4.2 suggest FRS is the best heuristics for many instances.

5.3.5 Computing Edge Order via Vertex Order

Now, we can obtain a vertex order by the algorithm presented in previous subsections. In the next step, we should transform it to an edge order. In Section 5.2.4, we have seen the edge ordering such that $|F_x| \leq |S_{j-1}| + 1$ holds. However, this ordering can be improved in terms of the entire frontier size.

If we fix the vertex order, the frontier transitions are uniquely determined as shown in Figure 5.3. Here, each vertex v has an interval in the frontier transition such that v is added to a frontier and v is removed from a frontier. This can be examined by the relation between frontiers and bags of path decomposition.

A key idea is that edge $e = \{u, v\}$ can be processed at any time in the intersection of the intervals of u and v. We use edge e at the minimum frontier in the valid interval. This reduces the entire frontier size since each edge is assigned into a smaller frontier than the frontier of the naïve ordering.

5.4 Experimental Results

We conducted experiments to evaluate our algorithm. All algorithms are implemented in C++ with g++ 4.9.3. We also use the TdZdd library [40] to implement frontier-based

[3] However, it must be greater than 0; $N(v) \cap S = \phi$ means v is not in the frontier.

Table 5.1. Comparing edge-ordering algorithms: naïve ordering v.s. proposed ordering.

	max frontier size	average frontier size					
	#improved instances	diff: naïve−proposed			ratio: naïve/proposed		
		average	min	max	average	min	max
Rome	63	0.152	0.065	0.346	1.019	1.007	1.038
VSPLIB	44	0.868	0.000	9.466	1.043	1.000	1.277

Table 5.2. Comparing linear time heuristics: previous algorithms and proposed algorithm. "#best" means the number of instances for which a corresponding method can compute the best frontier size. "ave. diff." means the average of the differences between the frontier size computed by a corresponding method and the best one.

		max frontier size					average frontier size				
		DFS	BFS	NDS	DLU	FRS	DFS	BFS	NDS	DLU	FRS
Rome	#best	0	0	1	5	137	0	0	0	4	136
	ave. diff.	7.78	7.99	7.11	4.14	0.02	4.63	4.71	4.40	2.48	0.01
VSPLIB	#best	5	22	9	21	47	4	21	4	13	42
	ave. diff.	38.42	8.33	19.37	10.11	4.25	21.13	4.19	10.86	5.10	2.35

search. We used a 3.20 GHz CPU machine with 64 GB memory.

We use two graph datasets: Rome Graph and VSPLIB. Rome Graph comprises road networks used as benchmark in [23]. We use 140 graphs with $n = 100$, $119 \leq m \leq 158$. VSPLIB is used as benchmark for the vertex separation problem in [27]. We use 73 instances of HB from VSPLIB. Graphs in VSPLIB have $24 \leq n \leq 960$ and $46 \leq m \leq 7442$.

5.4.1 Performance of Edge Ordering

We first show the results of the edge-ordering algorithm in Section 5.3.5. Here, we use vertex orders generated by FRS. Table 5.1 presents experimental results. Our edge-ordering strategy achieves reduction of the maximum frontier size by one in a half of instances and increases the maximum and average size in no instance. We thus use this algorithm to obtain an edge order from a vertex order.

5.4.2 Path Decomposition by Linear Time Heuristics

We next compare linear time heuristics, where a start vertex is fixed to a vertex with the minimum degree. Table 5.2 presents the experimental results. FRS is the best in terms of both of the maximum and average frontier size. Especially for the Rome Graph dataset, FRS is the best in almost all cases. Thus, we use FRS as linear time heuristics to determine the start vertices for the beam search.

Table 5.3. Comparing frontier sizes by BFS, BVNS, and beam search. "#best" and "ave. diff." mean the same as those in Table 5.2.

		max frontier size			average frontier size		
		BFS	BVNS	beam	BFS	BVNS	beam
Rome	#best	0	87	120	0	22	118
	ave. diff.	10.707	0.421	0.150	6.296	0.401	0.027
VSPLIB	#best	12	39	63	4	23	58
	ave. diff.	11.781	3.795	0.685	5.714	2.097	0.419

5.4.3 Path Decomposition by Meta-heuristics

We evaluate the performance of our beam search in terms of frontier size. We compare our algorithm with BFS, which is usually used in frontier-based search, and the BVNS in [27] with a time limit of 1000 seconds. We fix the beam width at $K = 5000$ and the number of start vertices at $L = 10$. Our algorithm runs in 0.64 to 787.59 seconds. Table 5.3 describes the experimental results.

Our algorithm achieves obtaining the best size for about 80% instances in both datasets. Furthermore, even if our algorithm fails to determine the best solution, it finds an order sufficiently close to the best, as shown in the average difference. On the other hand, BFS, commonly used heuristics for frontier-based search, seems not to be a sophisticated ordering for path decomposition.

5.4.4 Efficiency of Frontier-based search with Path Decomposition-based Ordering

Finally, we evaluate the impact of edge orders to frontier-based search. In this experiment, we construct a ZDD for all sub-forests with time limit 1000 seconds.

Table 5.4 provides the relation between the maximum frontier size F and the solved instances. The total number of solved instances by BFS ordering is really less than those determined by meta-heuristics, especially in Rome Graph. This shows that ordering based on a path decomposition is effective to improve the performance of frontier-based search. The boundary of the solvability of meta-heuristic methods seems to be $F = 11$ or 12. On the other hand, BFS sometimes constructs a ZDD up to $F = 16$ cases. This observation may be the key to improve our algorithm, but we have not revealed the details yet.

Table 5.5 summarizes the performance of frontier-based search using path decomposition-based ordering. The results show our method tends to be better than BVNS, especially in terms of the number of 2-fold or more improved instances. However, it seems to be a small improvement, whereas the maximum and average frontier size is significantly

Table 5.4. Relation between the maximum frontier size and the number of succeeded instances, where "in time" and "timeout" denote the number of instances for which frontier-based search runs within the time limit and exceeds the time limit, respectively.

		BFS								BVNS					beam			
		max frontier							sum	max frontier				sum	max frontier			sum
		≤ 11	12	13	14	15	16	$17 \leq$		≤ 10	11	12	$13 \leq$		≤ 11	12	$13 \leq$	
Rome	in time	0	0	2	1	3	0	0	6	87	29	7	0	123	121	4	0	125
	timeout	0	0	0	3	1	5	125	134	0	0	8	9	17	0	7	8	15
VSPLIB	in time	14	3	0	0	1	1	0	19	19	2	1	0	22	24	1	0	25
	timeout	0	1	0	0	0	0	53	54	0	1	0	50	51	0	1	47	48

Table 5.5. Comparing the performance of frontier-based search with BVNS and beam search. "average" is the average of results of instances for which both orderings constructed a ZDD. "#best" is the number of instances for which the ordering yields the best result. "#2-folds" is the number of instances for which the result of the ordering is more than twice as good as the other.

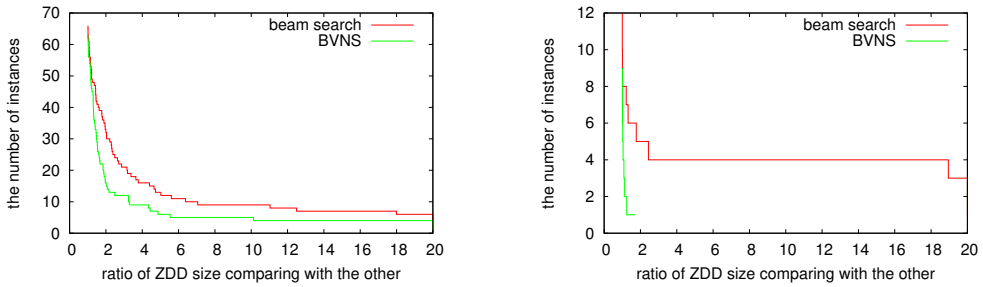
		runtime		non-reduced ZDD size		reduced ZDD size	
		BVNS	beam	BVNS	beam	BVNS	beam
Rome	average	35.81 s	32.68 s	140,496,029	129,300,247	11,542,286	11,010,751
	#best	63	66	60	69	63	66
	#2-folds	23	39	21	38	16	32
VSPLIB	average	41.11 s	25.30 s	135,520,389	83,321,337	22,976,532	14,510,650
	#best	10	15	10	12	9	12
	#2-folds	1	8	0	5	0	5

better than BVNS as shown in Section 5.4.3. This also indicates that there must be other factors than frontier size to improve frontier-based search.

Before the comparison between our method and BVNS, we notice the impact of the reduction of ZDDs. On the average, reduced ZDDs are more compact than non-reduced ones about 11 times in Rome Graph instances and 6 times in VSPLIB. This means DP calculation can be accelerated 6 to 11 times after ZDD reduction.

In 140 Rome Graph instances, ZDDs for 4 instances are constructed only by BVNS ordering, ZDDs for 6 instances are constructed only by beam search ordering, and 119 instances are constructed by both ordering. In 73 VSPLIB instances, ZDDs for 0 instances are constructed only by BVNS ordering, ZDDs for 3 instances are constructed only by beam search ordering, and 22 instances are constructed by both orderings.

The results of the averages and the number of the best looks like there is no big difference in our method and BVNS. We guess this is because BVNS also decreases the entire size by the evaluation function. On the other hand, we can find meaningful differences in the cases with the ratio is twice or more, i.e. there is significant improvement, especially in VSPLIB. Figure 5.4 illustrates details of the fact. In many cases, the two methods show close performance. But the number of meaningful improvements by our method is more than the one by BVNS.



(a) Rome Graph (b) VSPLIB

Figure 5.4. The number of instances with the ratio of reduced ZDD size comparing with the other method.

5.5 Concluding Remarks for Variable Ordering

We improved frontier-based search by focusing on edge ordering because it is robust to change constraints and affects not only runtime but also the size of the resulting ZDD. We proposed a meta-heuristic algorithm capable of determining a small frontier size by using the relation between frontiers and path decomposition.

Our algorithm has the ability to determine effective path decomposition in terms of both of the maximum and average bag size. Moreover, our algorithm achieves the construction of ZDDs for many instances for which the standard ordering BFS cannot construct ZDDs. Our algorithm also tends to reduce runtime and the size of a ZDD compared with the previous path decomposition method BVNS. On the other hand, the performance of frontier-based search with our ordering is sometimes worse than that with BVNS even for the case in which our algorithm yields more effective path decomposition than BVNS in terms of its bag sizes. Our future work aims to reveal the key factor responsible for this phenomenon for further improvement of frontier-based search.

Chapter 6

Conclusions and Open Problems

In this thesis, we aimed to enumerate all solutions of permutation problems. Since the number of solutions of permutation problems is factorial in the worst case, we use compressed data structures storing solutions and directly construct the data structure in order to avoid factorial time calculation. We focused on πDDs and Rot-πDDs as compressed data structures for permutation sets, because they are not only compact representation of permutations, also have rich operations to manipulate stored permutations.

In Chapter 3, we provided the algorithms for the two problems: reversible circuit debugging and cycle-type partition of a permutation set. For reversible circuit debugging, we proposed an efficient debugging algorithm without πDDs for erroneous circuits with a single error, and extended it to multiple errors by using πDDs to represent candidates of fixed circuits. This is the first algorithm that exactly debug erroneous circuits with multiple errors as far as the author knows. For cycle-type partition, we captured the nice relation between cycles and permutations in πDDs, and designed a construction algorithm by utilizing the advantage. Experimental results show that the algorithm is faster and less memory than the naïve method and the existing πDD construction method. These results show that there are many problems for which algorithms utilizing πDDs outperforms previous methods without decision diagrams, and ideas educing abilities of πDDs make the algorithms even faster.

In Chapter 4, we proposed a new decision diagram Rot-πDD, which is based on left-rotation decomposition of permutations. Furthermore, we applied Rot-πDDs to the three problems: enumeration of Eulerian trails, enumeration of topological orders, and enumeration of pattern-avoiding permutations. For Eulerian trails, we used dynamic programming on edge sets to directly construct Rot-πDDs, while DP-based construction for πDDs is not efficient. For topological orders, we indicated that the same dynamic programming approach as Eulerian trails also performs well for topological orders, and proved that the minimum path cover, a parameter of a directed graph, gives the tighter bound of the complexity. Moreover, we designed a new Rot-πDD operation to process

dynamic edge additions. For pattern-avoiding permutations, we reviewed a πDD-based algorithm, and Rot-πDDs can represent permutation sets in the middle of the algorithm with smaller sizes. This indeed contributes improvement of the practical performance of the algorithm as shown in the computational experiments. Since these applications reveal Rot-πDDs are also useful for permutation problems and which decision diagrams are preferable depends on problems, we also investigated the factors of problems in order to determine preferable decision diagrams for each problem. Preliminary experiments suggest that Rot-πDDs seem to have advantages if permutations in a given set have small disorder parameters except Exc.

In Chapter 5, we tackled a permutation problem in decision diagrams. We focused on the relation between path width and variable ordering for subgraph enumeration, and proposed a meta-heuristic search algorithm to optimize the path-width of a given graph. Experiments demonstrates that frontier based search and its resulting ZDD size are improved by using a variable order generated by the method.

The author now explores the following future directions:

1. The author guesses that there are many other problems for which algorithms with πDDs and Rot-πDDs outperforms existing methods. Of course, it is worth contributing the problems by applying decision diagrams to such problems. In addition, observing the properties of such problems and analyzing the performance of methods utilizing the properties are also useful to investigate the factor linked to the abilities of decision diagrams.

2. We may want to find an optimal permutation in a πDD or a Rot-πDD. For some functions, e.g. the sum of weights of transpositions or left-rotations, we can obtain an optimal one in time linear in the size of decision diagram, using Algorithm B in [50]. However, the author does not know ways to extract an optimal permutation for common weight-sum functions such as weights defined for pairs of a position and an element and, weights defined for relative orders between two elements. Efficient methods to extract optimal one for such functions are desired.

3. More deep theoretical analysis about compression ratio of πDDs and Rot-πDDs is needed. For compression of a single permutation, Barbay and Nabarro [5] states the relation between compression and adoptive sorting to disorder measures. The author guesses that compression ratio of πDDs and Rot-πDDS also theoretically depends some parameters, maybe some of disorder measures.

4. Conversely, we may be able to design new permutation decision diagrams based on other permutation decompositions. For example, reversals, reversing elements in each valid interval, can decompose permutations. For some of decompositions, we may be able to prove the theoretical size bound of decision diagrams with

respect to some permutation parameters. In general, we can decompose permutations in a permutation group by its generators. Thus, we can design general framework of permutation decision diagrams: generator-based πDDs. Here, generators must satisfy the following conditions:

- unique decomposability: for each permutation, we must define the corresponding generator sequence to make a decision diagram have a canonical form and path traversals on a decision diagram unique.
- ordered generators: generators must be ordered and applied without violation of the order. Moreover, each generator can be applied at most once. This condition corresponds to variable ordering of ZDDs.
- composition transformation: composition of two generators should has an equivalent composition form of other two generators, like transpositions and left-rotations. This is not needed for decision diagram structure, but for basic operations for manipulation of permutations in a decision diagram, like operations $\mathrm{Swap}(a,b)$ and $\mathrm{LeftRot}(a,b)$.

Acknowledgements

First of all, I would like to express my appreciation to my supervisor Professor Shin-ichi Minato. He always sincerely discussed with me and gives me new interesting insights for my research topics. His rich knowledges, including others than researches, help me in not only my research but also daily life. I also would like to thank Hiroki Arimura, the Professor of Laboratory of Information Knowledge Network, and Thomas Zeugmann, the Professor of Laboratory for Algorithmics, for their valuable comments and supports to write this thesis.

I would like to express my gratitude to my co-authors Takahisa Toda, Robert Wille, Norihito Yasuda, Nils Quetschlich, Shogo Takeuchi, Hiroyuki Hanada, Shuhei Denzumi, Hiroshi Aoki, Hirofumi Suzuki, Hana Ito, Fumito Takeuchi, and Kosuke Shiraishi. Especially, I would like to thank Takahisa Toda, Robert Wille, Norihito Yasuda, and Shuhei Denzumi for their fruitful discussion and advice. I also would like to thank Hirofumi Suzuki for providing implementations used in computational experiments in this thesis.

Fellowship of Japan Society for the Promotion of Science is gratefully acknowledged. A part of work in this thesis was supported by JSPS KAKENHI Grant Number 15J01665.

My deep appreciation goes to associate professor Ichigaku Takigawa, Dr. Charles Harold Jordan, and all other colleagues in my laboratories. I would be grateful to secretaries Sachiko Soma, Yukie Watanabe, and Yu Manabe for them invaluable support. Many friends have encouraged me. Finally, I am indebted to my parents and sisters for their support. Especially, my mother allowed me to go on to university and graduate school despite my father has passed away when I am a high school student. I definitely could not finish my degree and my thesis without a lot of her efforts and pains.

Bibliography

[1] Eyal Ackerman, Gill Barequet, and Ron Y. Pinter. A bijection between permutations and floorplans, and its applications. *Discrete Applied Mathematics*, 154(12):1674–1684, 2006.

[2] Michael Albert. Permlab: Software for permutation patterns. http://www.cs.otago.ac.nz/staffpriv/malbert/permlab.php, 2012.

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

[4] Eric Babson and Einar Steingrímsson. Generalized permutation patterns and a classification of the mahonian statistics. *Séminaire Lotharingien de Combinatoire*, 44, 2000.

[5] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[6] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52:3457–3467, 1995.

[7] Andrew M. Baxter and Lara K. Pudwell. Enumeration schemes for vincular patterns. *Discrete Mathematics*, 312(10):1699–1712, 2012.

[8] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 1108–1115. SIAM, 2009.

[9] Antoine Bérut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider, and Eric Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012.

[10] Hans L. Bodlaender, Ton Kloks, and Dieter Kratsch. Treewidth and pathwidth of permutation graphs. *SIAM Journal on Discrete Mathematics*, 8(4):606–616, 1995.

[11] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[12] Miklós Bóna. Exact enumeration of 1342-avoiding permutations: A close link with labeled trees and planar maps. *Journal of Combinatorial Theory, Series A*, 80(2):257–272, 1997.

[13] Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Information Processing Letters*, 65(5):277–283, 1998.

[14] Mireille Bousquet-Mélou, Anders Claesson, Mark Dukes, and Sergey Kitaev. (2+2)-free posets, ascent sequences and pattern avoiding permutations. *Journal of Combinatorial Theory, Series A*, 117(7):884–909, 2010.

[15] Graham R. Brightwell and Peter Winkler. Note on counting Eulerian circuits. Technical Report CDAM Research Report LSE-CDAM-2004-12, Centre for Discrete and Applicable Mathematics, London School of Economics, 2004.

[16] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[17] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[18] Russ Bubley and Martin Dyer. Faster random generation of linear extensions. *Discrete Mathematics*, 201(1-3):81–88, 1999.

[19] Hal Canary. Aztec diamonds and Baxter permutations. *The Electronic Journal of Combinatorics*, 17(1), 2010.

[20] Anupam Chattopadhyay, Soumajit Majumder, Chander Chandak, and Nahian Chowdhury. Constructive reversible logic synthesis for Boolean functions with special properties. In *Reversible Computation - 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings*, volume 8507 of *Lecture Notes in Computer Science*, pages 95–110. Springer, Cham, 2014.

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

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

[23] David Coudert, Dorian Mazauric, and Nicolas Nisse. Experimental evaluation of a branch and bound algorithm for computing pathwidth. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 46–58. Springer, Cham, 2014.

[24] Robert Cuykendall and David R. Andersen. Reversible optical computing circuits. *Optics Letters*, 12(7):542–544, 1987.

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

[26] James Donald and Niraj K. Jha. Reversible logic synthesis with Fredkin and Peres gates. *ACM Journal on Emerging Technologies in Computing Systems*, 4(1):2:1–2:19, 2008.

[27] Abraham Duarte, Laureano F. Escudero, Rafael Martí, Nenad Mladenovic, Juan José Pantrigo, and Jesús Sánchez-Oro. Variable neighborhood search for the vertex separation problem. *Computers & Operations Research*, 39(12):3247–3255, 2012.

[28] W. M. B. Dukes, M. F. Flanagan, T. Mansour, and V. Vajnovszki. Combinatorial Gray codes for classes of pattern avoiding permutations. *Theoretical Computer Science*, 396(1–3):35–49, 2008.

[29] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[30] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741.

[31] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.

[32] Stefan Frehse, Robert Wille, and Rolf Drechsler. Efficient simulation-based debugging of reversible logic. In *40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010*, pages 156–161. IEEE Computer Society, 2010.

[33] Éric Fusy. Bijective counting of involutive baxter permutations. *Fundamenta Informaticae*, 117(1–4):179–188, 2012.

[34] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.6*, 2016.

[35] Jens Gustedt. On the pathwidth of chordal graphs. *Discrete Applied Mathematics*, 45(3):233–248, 1993.

[36] Von Carl Hierholzer. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6:30–32, 1873.

[37] Xianlong Hong, Gang Huang, Yici Cai, Jiangchun Gu, Sheqin Dong, Chung-Kuan Cheng, and Jun Gu. Corner block list: An effective and efficient topological representation of non-slicing floorplan. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design*, pages 8–12. IEEE Computer Society, 2000.

[38] Takeru Inoue, Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato. Graphillion: software library for very large sets of labeled graphs. *International Journal on Software Tools for Technology Transfer*, 18(1):57–66, 2016.

[39] Takeru Inoue, Keiji Takano, Takayuki Watanabe, Jun Kawahara, Ryo Yoshinaka, Akihiro Kishimoto, Koji Tsuda, Shin-ichi Minato, and Yasuhiro Hayashi. Distribution loss minimization with guaranteed error bound. *IEEE Transactions on Smart Grid*, 5(1):102–111, 2014.

[40] Hiroaki Iwashita. TdZdd. https://github.com/kunisura/TdZdd, 2014.

[41] Hiroaki Iwashita and Shin-ichi Minato. Efficient top-down ZDD construction techniques using recursive specifications. Technical Report TCS-TR-A-13-69, Division of Computer Science, Hokkaido University, 2013.

[42] Hiroaki Iwashita, Yoshio Nakazawa, Jun Kawahara, Takeaki Uno, and Shin-ichi Minato. Efficient computation of the number of paths in a grid graph with minimal perfect hash functions. Technical Report TCS-TR-A-10-64, Division of Computer Science, Hokkaido University, 2013.

[43] Jean Christoph Jung, Stefan Frehse, Robert Wille, and Rolf Drechsler. Enhancing debugging of multiple missing control errors in reversible logic. In *the 20th ACM Great Lakes Symposium on VLSI 2010*, pages 465–470. ACM, 2010.

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

[45] Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. Technical Report TCS-TR-A-14-76, Division of Computer Science, Hokkaido University, 2014.

[46] Jun Kawahara, Toshiki Saitoh, Ryo Yoshinaka, and Shin-ichi Minato. Counting primitive sorting networks by πDDs. Technical Report TCS-TR-A-11-54, Division of Computer Science, Hokkaido University, 2011.

[47] Nancy G. Kinnersley. The vertex separation number of a graph equals its path-width. *Information Processing Letters*, 42(6):345–350, 1992.

[48] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968.

[49] Donald E. Knuth. Efficient representation of perm groups. *Combinatorica*, 11(1):33–43, 1991.

[50] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley, 2009.

[51] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[52] Thomas Lengauer. Black-white pebbles and graph separation. *Acta Informatica*, 16(4):465–475, 1981.

[53] 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.

[54] Darko Marinov and Radoš Radoičić. Counting 1324-avoiding permutations. *The Electronic Journal of Combinatorics*, 9(2), 2002.

[55] Dmitri Maslov, Gerhard W. Dueck, and D. Michael Miller. Techniques for the synthesis of reversible Toffoli networks. *ACM Transactions on Design Automation of Electronic Systems*, 12(4):42:1–42:28, 2007.

[56] Yasuko Matsui, Ryuhei Uehara, and Takeaki Uno. Enumeration of perfect sequences of chordal graph. In *Algorithms and Computation, 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, volume 5369 of *Lecture Notes in Computer Science*, pages 859–870. Springer, Berlin, Heidelberg, 2008.

[57] Brendan D. McKay and Robert W. Robinson. Asymptotic enumeration of Eulerian circuits in the complete graph. *Combinatorics, Probability and Computing*, 7(4):437–449, 1998.

[58] D. Michael Miller, Robert Wille, and Zahra Sasanian. Elementary quantum gate realizations for multiple-control Toffoli gates. In *41st IEEE International Symposium on Multiple-Valued Logic, ISMVL 2011, Tuusula, Finland, May 23-25, 2011*, pages 288–293. IEEE Computer Society, 2011.

[59] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *the 30th Design Automation Conference*, pages 272–277. ACM Press, 1993.

[60] Shin-ichi Minato. πDD: A New Decision Diagram for Efficient Problem Solving in Permutation Space. In *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 90–104. Springer, Berlin, Heidelberg, 2011.

[61] Yuchang Mo, Liudong Xing, Farong Zhong, Zhusheng Pan, and Zhongyu Chen. Choosing a heuristic and root node for edge ordering in BDD-based network reliability analysis. *Reliability Engineering & System Safety*, 131:83–93, 2014.

[62] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[63] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

[64] Akimitsu Ono and Shin-Ichi Nakano. Constant time generation of linear extensions. In *Fundamentals of Computation Theory, 15th International Symposium, FCT 2005, Lübeck, Germany, August 17-20, 2005, Proceedings*, volume 3623 of *Lecture Notes in Computer Science*, pages 445–453. Springer, Berlin, Heidelberg, 2005.

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

[66] Sheng-Lung Peng, Chin-Wen Ho, Tsan-sheng Hsu, Ming-Tat Ko, and Chuan Yi Tang. A linear-time algorithm for constructing an optimal node-search strategy of a tree. In *Computing and Combinatorics, 4th Annual International Conference,*

COCOON '98, Taipei, Taiwan, R.o.C., August 12-14, 1998, Proceedings, volume 1449 of *Lecture Notes in Computer Science*, pages 279–288. Springer, Berlin, Heidelberg, 1998.

[67] Asher Peres. Reversible logic and quantum computers. *Physical Review A*, 32(6):3266–3276, 1985.

[68] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *PNAS*, 98(17):9748–9753, 2001.

[69] Vaughan R. Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pages 268–277. ACM, 1973.

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

[71] Md Zamilur Rahman and Jacqueline E. Rice. Templates for positive and negative control Toffoli networks. In *Reversible Computation - 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings*, volume 8507 of *Lecture Notes in Computer Science*, pages 125–136. Springer, Cham, 2014.

[72] Kuntal Roy. Optimum gate ordering of CMOS logic gates using Euler path approach: Some insights and explanations. *Journal of Computing and Information Technology*, 15(1):85–92, 2007.

[73] Zahra Sasanian, Robert Wille, and D. Michael Miller. Realizing reversible circuits using a new class of quantum gates. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 36–41. ACM, 2012.

[74] Kyoko Sekine and Hiroshi Imai. Counting the number of paths in a graph via BDDs. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 80(4):682–688, 1997.

[75] Charles C. Sims. Computational methods in the study of permutation groups. In *Computational problems in abstract algebra*, pages 169–183. Pergamon, 1970.

[76] Zvezdelina E. Stankova. Forbidden subsequences. *Discrete Mathematics*, 132(1–3):291–316, 1994.

[77] Laura Tague, Mathias Soeken, Shin-ichi Minato, and Rolf Drechsler. Debugging of reversible circuits using πDDs. In *43rd IEEE International Symposium on*

Multiple-Valued Logic, ISMVL 2013, pages 316–321. IEEE Computer Society, 2013.

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

[79] Tommaso Toffoli. Reversible computing. In *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 632–644. Springer, Berlin, Heidelberg, 1980.

[80] T. van Aardenne-Ehrenfest and N. G. de Bruijn. Circuits and trees in oriented linear graphs. In *Classic Papers in Combinatorics*, pages 149–163. Birkhäuser Boston, 1987.

[81] Julian West. Sorting twice through a stack. *Theoretical Computer Science*, 117(1–2):303–313, 1993.

[82] Herbert S. Wilf. The patterns of permutations. *Discrete Mathematics*, 257(2–3):575–583, 2002.

[83] Robert Wille, Daniel Große, Gerhard W. Dueck, and Rolf Drechsler. Reversible logic synthesis with output permutation. In *The 22nd International Conference on VLSI Design*, pages 189–194. IEEE Computer Society, 2009.

[84] Robert Wille, Daniel Große, Stefan Frehse, Gerhard W. Dueck, and Rolf Drechsler. Debugging of Toffoli networks. In *Design, Automation and Test in Europe, DATE 2009*, pages 1284–1289. IEEE, 2009.

[85] Robert Wille, Mathias Soeken, Christian Otterstedt, and Rolf Drechsler. Improving the mapping of reversible circuits to quantum circuits using multiple target lines. In *18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, pages 145–150. IEEE, 2013.

[86] S. Gill Williamson. *Pólya Counting Theory : Combinatorics for Computer Science*. CreateSpace Independent Publishing Platform, 2012.

[87] Norihiro Yamada and Shin-ichi Minato. A πDD-based method for generating conjugacy classes of permutation groups. Technical Report TCS-TR-A-12-56, Division of Computer Science, Hokkaido University, 2012.

[88] Bo Yao, Hongyu Chen, Chung-Kuan Cheng, and Ronald Graham. Floorplan representations: Complexity and connections. *ACM Transactions on Design Automation of Electronic Systems*, 8(1):55–80, 2003.

[89] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of Numberlink and Slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.

[90] 菊地洋右. オイラー路の列挙. **情報処理学会 研究報告アルゴリズム** *(AL)*, 2010(7):1–4, 2010.