𝕋ℂ𝕊-𝕋ℝ-𝔹-**14**-9

𝕋ℂ𝕊 Technical Report

Master's Thesis: Generating PiDDs for Indexing Permutation Classes with Given Permutation Patterns

by

Yuma Inoue

Division of Computer Science

Report Series B

February 25, 2014

Hokkaido University
Graduate School of
Information Science and Technology

Email: yuma@mx-alg.ist.hokudai.ac.jp

Phone: +81-011-706-7681

Fax: +81-011-706-7681

Master's Thesis
平成 25 年度　修士学位論文

Generating PiDDs for Indexing Permutation Classes with Given Permutation Patterns

πDD を用いたパターンに基づく順列クラスの列挙索引化

Yuma Inoue

井上　祐馬

Laboratory for Algorithmics, Division of Computer Science
Graduate School of Information Science and Technology
Hokkaido University

北海道大学　大学院情報科学研究科
コンピュータサイエンス専攻　アルゴリズム研究室

February 25, 2014

Contents

1 Introduction **4**

1.1 Background 4

1.2 Contributions 5

1.3 Related Work 5

1.4 Thesis Structure 6

2 Preliminaries **7**

2.1 Permutations 7

2.2 Permutation Patterns 8

2.3 πDDs 9

2.3.1 ZDDs 9

2.3.2 πDDs 11

2.4 Multiset of Permutations 12

2.5 πDD Vectors 12

3 Main Results **16**

3.1 Generation of Classical Pattern-Containing Permutations 16

3.1.1 Construction of $\mathbb{B}$ 18

3.1.2 Construction of $\mathbb{C}$ 18

3.1.3 Construction of $\mathbb{A}$ 19

3.2 Generation of Vincular Pattern-Containing Permutations 20

3.3 Generation of Bivincular Pattern-Containing Permutations 21

3.4 Generation of Pattern-Avoiding Permutations 21

3.4.1 Generation Algorithm for all n-Permutations 22

3.4.2 Summary of Generation Algorithm for $Av_n(\sigma)$ 23

3.5 Generation of Permutations with Pattern Occurrence Counts 24

3.5.1 Generation Algorithm for $\mathbf{P}_{n,\sigma}$ 24

3.5.2 Generation Algorithm for $F_n^{(k)}(\sigma)$ 24

3.5.3 Summary of Generation Algorithm for $F_n^{(k)}(\sigma)$ 25

4 Experimental Results **26**

4.1 Results for Classical Pattern-Avoiding Permutations 27

4.2 Results for Vincular and Bivincular Patterns 28

4.3 Results for Permutations with Pattern Occurrence Counts 30

5 Conclusion **33**

Summary

Permutation is a basic concept in elementary combinatorics and discrete mathematics. Permutations appear in various problems such as sorting, ordering, matching, coding and many other real-life situations. Permutations are also important in group theory since it corresponds to a bijective function and generates symmetric groups.

Permutation pattern is an important topic about permutations in which many researchers are interested. A permutation pattern *occurs* in a permutation if there is a subsequence of the permutation with the same relative order as the pattern. Otherwise, a permutation *avoids* a permutation pattern. Permutation patterns are related to practical and abstract mathematical problems and can provide simple representations for such problems. For example, some *floorplans*, which are used for optimizing very-large-scale integration (VLSI) circuit designs, can be encoded as pattern-avoiding permutations. On the other hand, *strong Wilf-equivalence* has been proposed for analysis of permutation classes with permutation patterns. The generation of permutation classes with given permutation patterns is therefore an important topic in efficient VLSI design and mathematical analysis of permutation patterns.

In this thesis, we present an algorithm for generating pattern-avoiding permutations, and extend this algorithm beyond classical patterns to generalized patterns with more restrictions. Moreover, we present an algorithm for generating permutations in which a pattern σ occurs exactly k times. Our approach is based on *Permutation Decision Diagrams* (*PiDD*s, or π*DD*s), a data structure which can represent permutation sets compactly and support various set operations. We demonstrate the efficiency of our algorithms by computational experiments.

Chapter 1

Introduction

Permutation is a mathematical concept, which appears in various mathematical and practical problems. For example, sorting, ordering, matching, coding, and so on can be described as permutation problems. Permutations also appear in group theory. Permutations correspond to a bijective function between two groups and generates symmetric groups.

Permutation pattern is the important topic in the research area of permutations. A permutation pattern σ occurs in a permutation π if there is a subsequence in π which is order isomorphic to σ. Two numerical sequences $a = a_1a_2\dots a_n$ and $b = b_1b_2\dots b_m$ are *order isomorphic* if a and b have the same length and satisfy $a_i < a_j$ if and only if $b_i < b_j$ for all i, j. Conversely, π avoids σ if σ does not occur in π. Permutations that avoid a pattern σ are called *σ-avoiding* permutations.

1.1 Background

Research on pattern-avoiding permutations dates back to *stack sort*, which was proposed by Knuth in [13]. 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 [22], and the double-ended queue sorting problem [18], have been considered, and pattern-avoiding permutations were developed in that context.

After pattern-avoiding permutations were first studied, many researchers worked on computing the number of permutations that avoid given patterns. For example, 1342-avoiding permutations have been enumerated by a mathematical approach [5], and 1324-avoiding permutations have been counted by computer programs [15]. Moreover, the relations between classes of pattern-avoiding permutations have also been examined. Two patterns σ_1 and σ_2 are *Wilf-equivalent* if the number of n-permutations which avoid σ_1 is same as the number of n-permutations for σ_2 for all positive integers n. In [20], the nontrivial Wilf-equivalence between 4132-avoiding and 3142-avoiding was discovered. More generally, two patterns σ_1 and σ_2 are *strongly Wilf-equivalent* [11] if $|F_n^{(k)}(\sigma_1)| =$

$|F_n^{(k)}(\sigma_2)|$ for all non-negative integers n and k, where $|S|$ denotes the cardinality of a set S and $F_n^{(k)}(\sigma)$ denotes the set of permutations of length n in which σ occurs exactly k times. Unfortunately, few results are known about strong Wilf-equivalence, as stated in the survey [21]. The generation of permutation classes with pattern occurrence counts can contribute to not only discoveries of new strongly Wilf-equivalent classes, but also identifications of bijective functions between such classes.

Relations between pattern-avoiding permutations and mathematical problems have also been studied actively [8, 10]. In particular, Yao et al. [24] discovered a bijection between *mosaic floorplans* and *Baxter permutations*, which are generalized pattern-avoiding permutations, and Ackerman et al. [1] proposed a simple encoding and decoding between these. A floorplan is a topological partition of a rectangle into multiple rectangles, and have practical applications in areas such as VLSI design. Mosaic floorplans are a subclass of floorplans. Bijections between other pattern-avoiding permutations and other floorplan classes are also discussed in [1, 19]. Storing all pattern-avoiding permutations in 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 could contribute to solving practical problems.

1.2 Contributions

In this thesis, we provide a practically efficient algorithm for generating pattern-avoiding permutations. Furthermore, we extend our algorithm to handle some generalized patterns, such as vincular and bivincular patterns. We also present an algorithm for generating $F_n^{(k)}(\sigma)$ based on the above algorithms. Using experiments, we measure the performance of our algorithms and compare with a naive method. In the experiments, we find some patterns that are potentialy strongly Wilf-equivalent.

Our algorithms are based on *Permutation Decision Diagrams* (*PiDD*s, or *πDDs*), a data structure for compactly representating sets of permutations [17]. These diagrams not only achieve high compression of sets of permutations, but also support rich algebraic set operations such as union and intersection. The computation time of these operations depends on the size of the πDDs and not on the number of permutations they represent. If the πDD is small, computation is fast independently of the number of permutations. This is a key benefit of our algorithms given that most previous generation algorithms focused on polynomial-delay algorithms, whose overall time complexity depends on the number of solutions.

1.3 Related Work

Wilf [23] provided an amortized polynomial-delay algorithm that generates all permutations avoiding the identity pattern (i.e. $12\dots n$), and posed the question about the complexity of generation for other patterns. Bose et al. [6] proved that the enumeration problem for pattern-avoiding permutations is #P-complete. Theoretically efficient generation algorithms for some particular patterns have been proposed [9]. On the other hand, as an instance of practical results, Albert [2] develops *PermLab*, which is software enumerating and listing pattern-avoiding permutations.

Our goal differs from the above results from the following viewpoints.

- Implicit generation:
  For applications such as Wilf-equivalence and floorplan databases, it is sufficient to store some information equivalent to all pattern-avoiding permutations. We do not have to list them explicitly.
- General patterns:
  We want to analyze as many patterns as possible.
- Practical efficiency:
  Theoretically fast algorithms are important. However, for practical applications, experimentally fast algorithms are also required.
- Enumeration with occurrences:
  The generation problem for pattern-avoiding permutations, i.e. $F_n^{(0)}(\sigma)$, has been studied extensively. However, as far as the author knows, no algorithm generating $F_n^{(k)}(\sigma)$ for any non-negative integer k has been proposed.

1.4 Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 introduces permutation patterns and πDDs. Chapter 3 presents our generation algorithms. Chapter 4 shows experimental results. The results include some candidates of strongly Wilf-equivalent classes. Chapter 5 concludes this thesis.

Chapter 2

Preliminaries

In this chapter, we give some notations and definitions for permutation patterns. We also define some generalized patterns, which have more restrictions besides order isomorphism. We also introduce πDDs and its extension for multisets of permutations. Our generation algorithms, which will be described in Chapter 3, are based on these data structures.

2.1 Permutations

A permutation of length n (n-permutation for brevity) is a bijection from $\{1,2,...,n\}$ to itself. Hereafter, we write permutations in one line as $\pi = \pi(1)\pi(2)\ldots\pi(n)$, and denote $\pi_i = \pi(i)$. The *k-prefix* of π denotes the first k numbers in π, and the *k-suffix* of π denotes the last k numbers in π. For example, $\pi = 4312$ is a 4-permutation, $\pi_3 = 1$, and the 2-prefix of π is 43. An element π_i in π is *fixed* if $\pi_i = i$ holds.

Multiplication over permutations x and y is defined as $x \cdot y = y_{x_1}y_{x_2}\ldots y_{x_n}$, which is applying y after x. Note that the leftmost permutation is applied first. For example, let $x = 45213$ and $y = 41352$, then $x \cdot y = 52143$ (see Figure 2.1). Note that multiplication over permutations is non-commutative. We denote by e_n the identity permutation of length n, where e_n satisfies $e_n(i) = i$ for each $1 \leq i \leq n$.

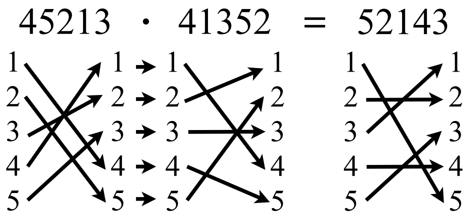


Figure 2.1. An example of multiplication of permutations.

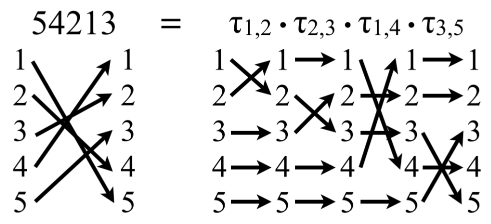


Figure 2.2. Decomposition of 54213 into transpositions.

In this thesis, we use the multiplication $x \cdot y$ in order to permute the numbers in y according to the order of numbers in x, that is, y_i is placed in the kth position with $x_k = i$. We call this operation the rearrangement of y according to x. For example, let $r = 54321$, which is the reverse of e_5, and $\pi = \pi_1\pi_2\pi_3\pi_4\pi_5$. Then we obtain $r \cdot \pi = \pi_5\pi_4\pi_3\pi_2\pi_1$, which is the reverse of π.

A *transposition* is a permutation that exchanges only two elements. More precisely, a transposition $\tau_{i,j}$ is a permutation such that $\tau_{i,j}(i) = j$, $\tau_{i,j}(j) = i$, and $\tau_{i,j}(k) = k$ for all other numbers k. Any n-permutation can be uniquely represented as the product of at most $n-1$ transpositions based on a straight selection sorting algorithm [14]. This algorithm repeatedly swaps the value k and the kth element from right to left. For example, consider the decomposition of the permutation 54213 into a product of transpositions (Figure 2.2). The 5th element of 54213 is 3, hence we exchange 5 and 3, and obtain $54213 = 34215 \cdot \tau_{3,5}$. Since the 4th element of 34215 is 1, we then obtain $54213 = 31245 \cdot \tau_{1,4} \cdot \tau_{3,5}$. Repeating this procedure, we finally obtain $54213 = \tau_{1,2} \cdot \tau_{2,3} \cdot \tau_{1,4} \cdot \tau_{3,5}$.

2.2 Permutation Patterns

A permutation π *contains* a permutation σ, or σ *occurs* in π 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. In other words, let l be the length of σ. Then, π contains σ if there are indexes $1 \leq i_1 < i_2 < \ldots < i_l \leq n$ such that $\pi_{i_x} < \pi_{i_y}$ if and only if $\sigma_x < \sigma_y$, for all pairs of x and y. Such σ is called a *pattern*. For example, the permutation 4213 contains the pattern 312 because 423 and 413 are order isomorphic to the pattern. Conversely, π *avoids* σ if π does not contain σ.

The above pattern is called a *classical pattern* because some generalizations have been proposed. For example, the *vincular pattern*, which is also called the *generalized pattern*, is a well-known generalization [3]. While the defining restriction in classical patterns is order isomorphism, vincular patterns additionally have another restriction: adjacency of element positions in the permutation. We use the underline notation to represent adjacencies. If the ith and the $(i+1)$th elements are underlined, the corresponding numbers in the permutation must be adjacent. For example, we consider the permutation 4213 and the vincular pattern $3\underline{12}$. Both 423 and 413 are order isomorphic to 312, but 423 does not match $3\underline{12}$ because the second and third elements are not adjacent in the permutation.

In contrast, 413 matches the pattern because 1 and 3 are adjacent in 4213. Thus, 4213 contains $3\underline{12}$.

Vincular patterns have been extended to *bivincular patterns* [7]. A bivincular pattern is restricted by adjacency of positions and additionally by consecutiveness of values. We use a two-line form with bars and underlines to represent bivincular patterns. The first row represents consecutiveness and an identical order, and the second row represents adjacencies and a relative order. For example, the bivincular pattern $\genfrac{}{}{0pt}{}{\overline{12}3}{3\underline{12}}$ 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 permutation, and the relative order must match 312. Thus, the permutation 4213 avoids $\genfrac{}{}{0pt}{}{\overline{12}3}{3\underline{12}}$. Indeed, both subsequences 423 and 413 are order isomorphic to 312 but 423 does not match the bivincular pattern because 2 and 3 are not adjacent in the permutation, and 413 also does not match the bivincular pattern because 1 and 3 are not consecutive.

The *pattern occurrence count* of σ in π is the number of distinct subsequences in the permutation π which are order isomorphic to the pattern σ. For example, the pattern occurrence count of 312 in 4213 is 2 as described above. Hereafter, $F_n^{(k)}(\sigma)$ denotes the set of n-permutations in which σ occurs exactly k times. Two patterns σ_1 and σ_2 are *Wilf-equivalent* if they have the same number of n-permutations which avoid the patterns, i.e., $|F_n^{(0)}(\sigma_1)| = |F_n^{(0)}(\sigma_2)|$ holds for all positive integers n. More generally, two patterns σ_1 and σ_2 are *strongly Wilf-equivalent* if $|F_n^{(k)}(\sigma_1)| = |F_n^{(k)}(\sigma_2)|$ holds for all non-negative integers n and k.

The problem considered in this thesis can be stated as follows: given a positive integer n and a pattern σ, implicitly generate $F_n^{(k)}(\sigma)$ for all non-negative integers k.

2.3 πDDs

A Permutation Decision Diagram (PiDD, or πDD) [17] is a data structure which canonically represents and efficiently manipulates a set of n-permutations. The efficiency of our algorithms is based on the compact representation and rich set operations of πDDs. The structure of πDDs is based on *Zero-suppressed Binary Decision Diagrams* (*ZDDs*) [16], which are decision diagrams for sets of combinations (families of sets).

2.3.1 ZDDs

A ZDD is derived by reducing a binary decision tree. Figure 2.3 shows the ZDD for the family of sets $\{\{a,b\},\{a,c\},\{c\}\}$. A ZDD has five components: internal nodes with an item label, 0-edges, 1-edges, the 0-terminal node, and the 1-terminal node.

A ZDD forms a connected DAG (Directed Acyclic Graph) with a single root node. Each node except the two terminal nodes has exactly one 0-edge and one 1-edge. Each

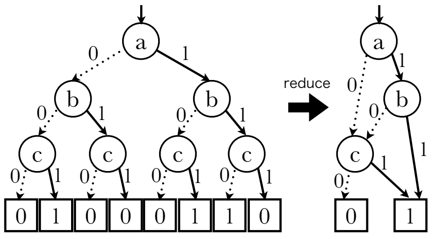


Figure 2.3. A binary decision tree and a ZDD for a set of combinations.

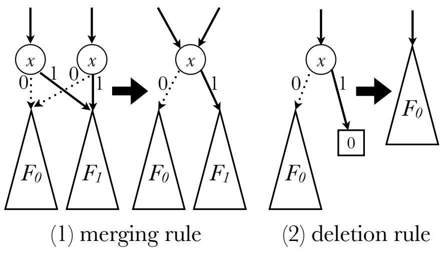


Figure 2.4. Two reduction rules on ZDDs.

path represents a combination of items: if a 1-edge originates from a node with label *x*, the combination contains *x*, while a 0-edge from *x* means that the combination excludes *x*. If a path reaches the 1-terminal node, the combination represented by the path is in the set represented by the ZDD. On the other hand, if a path reaches the 0-terminal node, the combination is not in the set. For a node *N*, we call the subgraph pointed to by its 0-edge the *left subgraph* of *N* and the subgraph pointed to by its 1-edge the *right subgraph* of *N* for convenience. Then, we can consider that the 0-terminal node represents an empty set, the 1-terminal node represents a singleton of a combination which has no item, and a node with label *x* represents the union of the left subgraph and the right subgraph whose each combination added the item *x* to.

A ZDD is canonical, and often compact representation if we fix the order of the items and apply the following two reduction rules:

(1) Merging rule: Share all nodes which have the same left subgraphs, the same right subgraphs, and the same labels

(2) Deletion rule: Delete all nodes whose 1-edge point directly to the 0-terminal node.

These rules are illustrated in Figure 2.4. In the worst-case scenario, the size of a ZDD (i.e. the number of nodes in a ZDD) can grow exponentially with respect to the number of items. In many practical cases, though, ZDDs provide efficient compression.

In addition, ZDDs support efficient set operations such as union, intersection, and set difference. Since these operations are realized by recursive algorithms with hash table techniques, the computation time of these operations depends on the number of nodes in the ZDDs, not on the cardinality of the sets they represent.

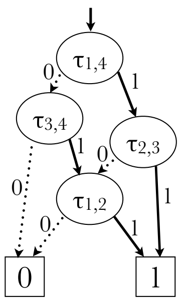


Figure 2.5. The πDD for $\{2143, 2431, 4321\} = \{\tau_{1,2} \cdot \tau_{3,4}, \tau_{1,2} \cdot \tau_{1,4}, \tau_{2,3} \cdot \tau_{1,4}\}$.

2.3.2 πDDs

We introduce πDDs, which represent a set of n-permutations, by deriving them from a ZDD. As shown in Section 2.1, any permutation can be uniquely decomposed into a product 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 $\tau_{x_1,y_1} < \tau_{x_2,y_2}$ is true if $y_1 > y_2$ holds, or $y_1 = y_2$ and $x_1 < x_2$ holds, and use this order as the fixed order in a πDD. In πDDs, the 0-terminal node represents an empty set and the 1-terminal node represents a singleton of the identity permutation e_n. We obtain compact and canonical πDDs by applying the above two reduction rules in the same way to ZDDs, and set operations are also available. Figure 2.5 shows an example of a πDD.

Table 2.1 shows the πDD operations used in this thesis. Here the Cartesian product of πDDs differs from the usual Cartesian product of sets: the set of multiplications for all pairs of permutations in one permutation set and those in the other permutation set. While the union and set difference operations are available like ZDDs, the swap and Cartesian product operations are unique to πDDs. In particular, the Cartesian product is very useful because multiplications 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.

P.Top	return the transposition $\tau_{x,y}$ by which the root node is labeled.
$P \cap Q$	return intersection $\{\pi \mid \pi \in P$ and $\pi \in Q\}$.
$P \cup Q$	return union $\{\pi \mid \pi \in P$ or $\pi \in Q\}$.
$P \setminus Q$	return difference $\{\pi \mid \pi \in P$ and $\pi \notin Q\}$.
$P \cdot \tau_{x,y}$	return swap $\{\pi \cdot \tau_{x,y} \mid \pi \in P\}$.
$P \times Q$	return Cartesian product $\{\alpha \cdot \beta \mid \alpha \in P$ and $\beta \in Q\}$.

2.4 Multiset of Permutations

In this section, we introduce some definitions and notations for multisets of permutations. Let $\mathbf{P} = \langle P, f \rangle$ be a multiset over P, where P is a permutation set and $f : P \to \mathbb{N}$ is a function. Here, $f(\pi)$ is the multiplicity of permutation π and we use the notation $\mathbf{P}(\pi)$ instead of $f(\pi)$ for convenience. We define $\mathbf{P} \uplus \mathbf{Q}$ as the *multiset sum* of two multisets $\mathbf{P}$ and $\mathbf{Q}$, where $(\mathbf{P} \uplus \mathbf{Q})(\pi) = \mathbf{P}(\pi) + \mathbf{Q}(\pi)$ holds for all permutations π. *Scalar multiplication* of an integer k and a multiset $\mathbf{P}$ is defined by $k \cdot \mathbf{P} = \langle P, k \cdot f \rangle$. Cartesian product of two multisets $\mathbf{P} = \langle P, f \rangle$ and $\mathbf{Q} = \langle Q, g \rangle$ is defined by $\mathbf{P} \times \mathbf{Q} = \langle P \times Q, h(\pi \cdot \pi') = f(\pi) \cdot g(\pi') \rangle$, where $\pi \in P$, $\pi' \in Q$, and $P \times Q$ means Cartesian product of permutation sets like that of πDDs.

A set of permutations weighted by pattern occurrence counts can be represented by a multiset. Given a positive integer n and a pattern σ, $\mathbf{P}_{n,\sigma} = \langle S_n, f_\sigma \rangle$ denotes the set of n-permutations weighted by the pattern occurrence count of σ, i.e., $f_\sigma(\pi)$ equals the pattern occurrence counts of σ in π. Our algorithm, which will be described in Section 3.5, first generates $\mathbf{P}_{n,\sigma}$, and then computes $F_n^{(k)}(\sigma) = \{\pi \mid \mathbf{P}_{n,\sigma}(\pi) = k\}$ from $\mathbf{P}_{n,\sigma}$.

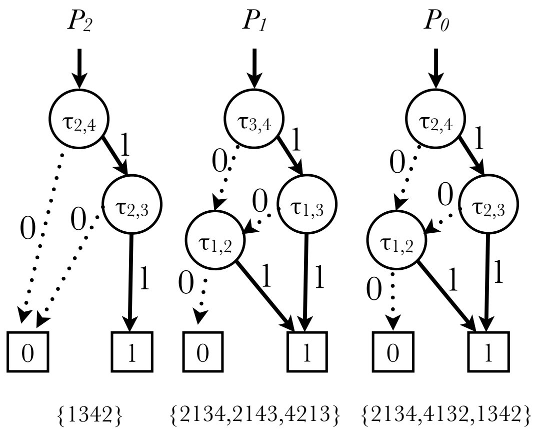
2.5 πDD Vectors

In this thesis, we want to handle multisets of permutations. But πDDs cannot represent multisets. To overcome this problem, we use *πDD vectors*, which were proposed in [12]. A πDD vector is a data structure for a multiset of permutations based on a binary representation using multiple πDDs. A πDD vector consists of an array of πDDs. Let M be the maximum multiplicity of a permutation in a given multiset. We denote a πDD vector by $\vec{P} = (P_0, P_1, \ldots, P_m)$, where each P_i is a πDD and $m = \lfloor \log M \rfloor$. Let $\vec{P}(\pi)$ be the multiplicity of π in $\vec{P}$. If a permutation π is in P_i, the ith bit of the binary representation of $\vec{P}(\pi)$ is 1, and otherwise 0. In other words, $\vec{P}(\pi) = \sum_{i=0}^{m} 2^i \cdot [\pi \in P_i]$ holds, where $[x]$ equals 1 if x is true, and otherwise 0. Figure 2.6 shows an example of a πDD vector. In actuality, πDDs in a πDD vector share common subgraphs as illustrated in [12]. Thus, more efficient compression can be expected.

πDD vectors have multiset operations. Table 2.2 shows some πDD vector operations which were used in this thesis. Most operations in Table 2.2 were proposed in [12]. Un-

Table 2.2. πDD operations on πDD vectors $\vec{P}$ and $\vec{Q}$.

$\vec{P} \uplus \vec{Q}$	Return multiset sum $\vec{R}$ such that $\vec{R}(\pi) = \vec{P}(\pi) + \vec{Q}(\pi)$ holds.
$\vec{P} \cdot \tau_{x,y}$	Return swap $\vec{P}' = (P_0 \cdot \tau_{x,y}, P_1 \cdot \tau_{x,y}, \ldots, P_m \cdot \tau_{x,y})$.
$\vec{P}$.numberof(π)	Return $\vec{P}(\pi)$.
$\vec{P} \times \vec{Q}$	Return Cartesian product $\vec{R}$ such that $\vec{R}(\alpha \cdot \beta) = \vec{P}(\alpha) \cdot \vec{Q}(\beta)$.



Permutation	(P_2,P_1,P_0)	Multiplicity
1342	(1,0,1)	5
2134	(0,1,1)	3
2143	(0,1,0)	2
4213	(0,1,0)	2
4132	(0,0,1)	1

Figure 2.6. An example of a πDD vector.

fortunately, however, the Cartesian product operation of πDD vectors has not ever been proposed. We introduce the Cartesian product operation.

Cartesian product operation between two πDDs P and Q was defined as follows ([17]):

$$P \times Q = (P \times Q^l) \cup ((P \times Q^r) \cdot \tau_{x,y}),$$

where Q^l and Q^r are the left and the right subgraphs of the root node of Q, respectively, and $\tau_{x,y}$ is the transposition associated with the root node of Q.

We extend Cartesian product to πDD vectors as follows:

$$\vec{P} \times \vec{Q} = (\vec{P} \times \vec{Q}^L) \uplus ((\vec{P} \times \vec{Q}^R) \cdot \tau_{x,y}),$$

where multiset sum $\uplus$ and swap $\tau_{x,y}$ are πDD vector operations which were already defined in [12] as shown in Table 2.2. However, some Q_i in the array of $\vec{Q}$ can have distinct transpositions in their root nodes. We need to define $\vec{Q}^L$ and $\vec{Q}^R$, and $\tau_{x,y}$ appropriately.

We use τ_{x_s,y_s} which is the smallest transposition in the root nodes of $Q_1, Q_2, \dots, Q_m$, where the order of transpositions is introduced in Section 2.3.2. For example, for the πDD vector illustrated in Figure 2.6, τ_{x_s,y_s} is $\tau_{2,4}$ because the transpositions on the root nodes of P_0, P_1, and P_2 are $\tau_{2,4}$, $\tau_{3,4}$, and $\tau_{2,4}$, respectively, and the smallest transposition among them is $\tau_{2,4}$. Here, we define $\vec{Q}^L$ and $\vec{Q}^R$ as follows:

$$\vec{Q}^L = (Q_0^L, Q_1^L, \dots Q_m^L), \text{where } Q_i^L = \begin{cases} Q_i \text{ if } Q_i.\text{Top} \neq \tau_{x_s,y_s}, \\ Q_i^l \text{ otherwise.} \end{cases}$$

$$\vec{Q}^R = (Q_0^R, Q_1^R, \dots Q_m^R), \text{where } Q_i^R = \begin{cases} \emptyset \text{ if } Q_i.\text{Top} \neq \tau_{x_s,y_s}, \\ Q_i^r \text{ otherwise.} \end{cases}$$

If the root node in Q_i is not labeled by $\tau_{x,y}$, it is equivalent to the πDD whose root node has the label $\tau_{x,y}$, Q_i as the left subgraph, and the 0-terminal node as the right subgraph. This implies correctness of the above definition.

We must also define the base case of the above recursion. The base case means every root node of Q_i is not labeled by a transposition, that is, each Q_i consists only of a terminal node. Since each Q_i consists only of the 0-terminal node or the 1-terminal node, i.e. an empty set or a singleton of e_n, respectively, $\vec{Q}(e_n) \geq 0$ and $\vec{Q}(\pi) = 0$ for the other permutations π hold. Then, $\vec{P} \times \vec{Q}$ equals the scalar multiplication $\vec{Q}(e_n) \cdot \vec{P}$. Here, we define $(k_l k_{l-1} \dots k_0)_2$ as a binary representation of non-negative integer k, where $l = \lfloor \log k \rfloor$. Algorithm 1 gives an algorithm for a scalar multiplication of a πDD vector based on the multiset sum of $2^i \cdot \vec{P} = (\underbrace{\emptyset, \dots, \emptyset}_{i}, P_0, \dots, P_m)$. In conclusion, Algorithm 2 shows how to compute the Cartesian product of πDD vectors.

Algorithm 1 Scalar multiplication of a non-negative integer $k = (k_l k_{l-1} \dots k_0)_2$ and a πDD vector $\vec{P} = (P_0, P_1, \dots, P_m)$.

```
$\pi$DD vector $\vec{R} \leftarrow (\emptyset)$
for $i = 0$ to $l$ do
    if $k_i = 1$ then
        $\vec{R} \leftarrow \vec{R} \uplus \vec{P}$
    end if
    $\vec{P} \leftarrow (\emptyset, P_0, P_1, \dots, P_{m+i})$
end for
return $\vec{R}$
```

Algorithm 2 Cartesian product of two πDD vectors $\vec{P} = (P_0, P_1, \ldots, P_{m'})$ and $\vec{Q} = (Q_0, Q_1, \ldots, Q_m)$.

```
{Calculate the smallest transposition.}
{We suppose the 0-terminal node is labeled by $\tau_{0,0}$ and the 1-terminal node is labeled
by $\tau_{1,1}$ for convenience.}
transposition $t \leftarrow \tau_{0,0}$
**for** $i = 0$ to $m$ **do**
  **if** $Q_i$.Top $< t$ **then**
    $t \leftarrow Q_i$.Top
  **end if**
**end for**

{Branch based on $t$.}
**if** $t = \tau_{0,0}$ **then**
  {Every $Q_i$ consists only of the 0-terminal node.}
  **return** $(\emptyset)$
**else if** $t = \tau_{1,1}$ **then**
  {Every $Q_i$ consists only of the 0-terminal node or the 1-terminal node.}
  **return** $\vec{Q}$.numberof$(e_n) \cdot \vec{P}$
**else**
  {There is at least one $Q_i$ whose root node is not a terminal node.}
  **for** $i = 0$ to $m$ **do**
    **if** $Q_i$.Top $= t$ **then**
      $Q_i^L \leftarrow Q_i^l$, $Q_i^R \leftarrow Q_i^r$
    **else**
      $Q_i^L \leftarrow Q_i$, $Q_i^R \leftarrow \emptyset$
    **end if**
  **end for**
  **return** $(\vec{P} \times \vec{Q}^L) \uplus ((\vec{P} \times \vec{Q}^R) \cdot t)$
**end if**
```

Chapter 3

Main Results

In this chapter, we gives generation algorithms for pattern-containing permutations, pattern-avoiding permutations, and permutations with pattern occurrence counts. We also present the extension for some general patterns.

3.1 Generation of Classical Pattern-Containing Permutations

Hereafter, unless otherwise noted, l denotes the length of a given pattern σ and $C_n(\sigma)$ denotes the set of n-permutations that contain σ. The algorithm for generating $C_n(\sigma)$ is sketched as follows. First, we generate all numerical sequences satisfying the following conditions:

- It is order isomorphic to σ.
- The elements in it are less than or equal to n.

The number of such numerical sequences is $\binom{n}{l}$. For example, if we suppose that $n = 5$ and $\sigma = 312$, then there are ten such numerical sequences: $312, 412, \ldots, 534$. Next, we generate all n-permutations such that the above numerical sequences appear as a subsequence. In order to do this, it is sufficient to embed the numerical sequences in sequences of length n without changing their order. For example, the numerical sequence 524 can be embedded in the following ways: $524{*}{*}, 52{*}4{*}, \ldots, {*}{*}524$. Here, we assign other positive integers to the $*$ positions in any order. Then, we obtain all n-permutations which have 524 as their subsequence: 52413, 52431, 52143, 52341, …, 13524, 31524. Since all order isomorphic sequences are embedded in all possible positions, this process generates $C_n(\sigma)$.

Replacing all numerical sequences in the above process by permutations, this process is described as the three steps as follows:

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

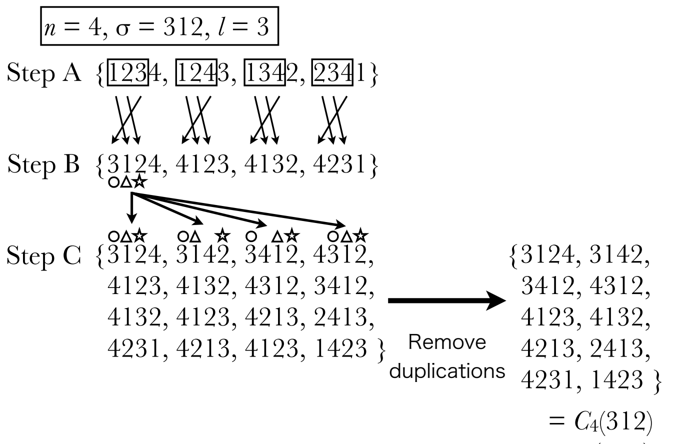


Figure 3.1. The process of generating $C_4(312)$.

B. Rearrange the l-prefix of each permutation which was generated in step A into the order isomorphic to σ,

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

Figure 3.1 shows the process of generating $C_4(312)$. Step A generates all $\binom{n}{l}$ combinations in the l-prefix of the permutations. Step B rearranges the l-prefix of each permutation into the numerical sequence order isomorphic to σ. All possible numerical sequences order isomorphic to σ appear in the l-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 $\binom{n}{l}$ subsequences in π exactly matches one of the numerical sequences order isomorphic to σ. They may be some duplications. But we do not have to consider the duplications when using πDDs because πDDs represent the set, not multiset.

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 2.3.2. Let $\mathbb{A}$ denote the πDD for permutations which are generated in step A, and let $\mathbb{B}$ and $\mathbb{C}$ 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 3.2). Note that the Cartesian products must be applied in the reverse order of the three steps we execute, that is, we first apply $\mathbb{B}$ to $\mathbb{A}$ and then apply $\mathbb{C}$ to the result of the first application.

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

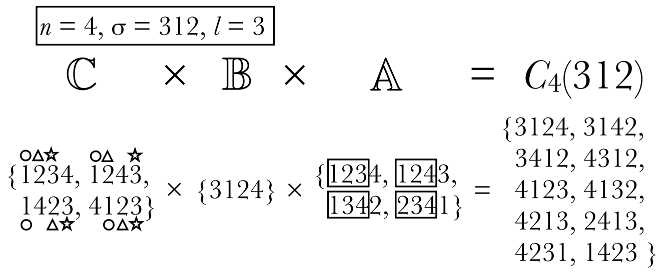


Figure 3.2. Cartesian product in generating $C_4(312)$.

3.1.1 Construction of $\mathbb{B}$

In order to rearrange the l-prefix of all permutations in $\mathbb{A}$ into the order isomorphic to σ, we define $\mathbb{B}$ to be the πDD consisting only of the permutation whose l-prefix is σ and $(n-l)$-suffix is fixed. To construct this πDD, we first decompose σ into a product of transpositions as given in Section 2.1. The πDD forms one path based on this decomposition, and can be easily constructed in a bottom-up fashion.

3.1.2 Construction of $\mathbb{C}$

We define $\mathbb{C}$ to be the πDD for the set of n-permutations π such that there are l indixes $1 \le p_1 < p_2 < \ldots < p_l \le n$ with $\pi_{p_i} = i$. This means that each permutation in $\mathbb{C}$ must have the numerical sequence $12\ldots l$ as a 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 it must repeat constructions and union operations $\binom{n}{l}$ times.

Our idea to reduce the number of operations is based on Pascal's rule, which is the combinatorial identity $\binom{n}{l} = \binom{n-1}{l} + \binom{n-1}{l-1}$. Let $Pos_{i,j}$ be the set of all n-permutations π containing at least one subsequence $\pi_{l_1}\pi_{l_2}\ldots\pi_{l_j}$ satisfying the following two conditions:

1. $1 \le l_1 < l_2 < \ldots < l_j \le i$.
2. $\pi_{l_1}\pi_{l_2}\ldots\pi_{l_j} = 12\ldots j$,

It is obvious that $\mathbb{C}$ is the πDD for $Pos_{n,l}$. If we can calculate $Pos_{i,j}$ using $Pos_{i-1,j}$ and $Pos_{i-1,j-1}$ as in the identity above, we can obtain $\mathbb{C}$ with only $O(nl)$ operations. In order to make this idea work, we restrict $Pos_{i,j}$ with the additional condition as follows:

3. For each $i+1 \le x \le n$, 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, if $\mathbb{P}_{i,j}$ denotes the πDD for $Pos_{i,j}$, this is achieved by $\mathbb{P}_{i-1,j-1} \cdot \tau_{i,j}$ because the ith element is i from the third condition and j is not assigned yet. Thus, $\mathbb{P}_{i,j} = \mathbb{P}_{i-1,j} \cup \mathbb{P}_{i-1,j-1} \cdot \tau_{i,j}$ holds. The dynamic programming for this recursion is encoded in Algorithm 3.

Algorithm 3 Construct $\mathbb{C}$.

```
$\mathbb{P}_{0,0} \leftarrow \pi$DD for $\{e_n\}$
for $i = 1$ to $n$ do
    for $j = 0$ to $l$ do
        if $j > 0$ then
            $\mathbb{P}_{i,j} \leftarrow \mathbb{P}_{i-1,j} \cup \mathbb{P}_{i-1,j-1} \cdot \tau_{i,j}$
        else
            $\mathbb{P}_{i,j} \leftarrow \mathbb{P}_{i-1,j}$
        end if
    end for
end for
return $\mathbb{P}_{n,l}$
```

3.1.3 Construction of $\mathbb{A}$

As shown in Section 3.1, $\mathbb{A}$ is the πDD for the set of all n-permutations whose l-prefix is ordered in increasing order. More precisely, a permutation π in $\mathbb{A}$ satisfies $1 \leq \pi_1 < \pi_2 < \ldots < \pi_l \leq n$.

We can obtain $\mathbb{A}$ by repeating assignments in a similar way to the construction of $\mathbb{C}$. However, this does not work if we assign elements in increasing order. In order to assign v to pth position by $\tau_{p,v}$, p must be fixed to pth position. But given that $p \leq v$, other transpositions could result in p being in another position. In the construction of $\mathbb{P}_{p,v}$, there is no such problem because after $\tau_{p,v}$ assigns v to pth position, there are no further assignments to vth position due to $v \leq p$. Therefore, we reverse the order of assignments in order to fix the position of elements which will be used in the future. Let $Inc_{i,j}$ be the set of all n-permutations satisfying the following conditions:

1. $i+1 \leq \pi_{j+1} < \pi_{j+2} < \ldots < \pi_l \leq n$,
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 $(l-j)$-suffix of the l-prefix of π is already used and the j-prefix of the l-prefix of π is fixed. We execute the constructions from $Inc_{n,l} = \{e_n\}$ to $Inc_{0,0}$. If $\mathbb{I}_{i,j}$ denotes the πDD for $Inc_{i,j}$, then

$\mathbb{I}_{i,j} = \mathbb{I}_{i+1,j} \cup \mathbb{I}_{i+1,j+1} \cdot \tau_{i+1,j+1}$ holds because $\mathbb{I}_{i,j}$ can be partitioned into the set including π with $\pi_{j+1} = i+1$, which is $\mathbb{I}_{i+1,j+1} \cdot \tau_{i+1,j+1}$ 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 $(n-l)$-suffix of each permutation in $\mathbb{I}_{0,0}$ is in one fixed order. We must generate all orders of the $(n-l)$-suffix of each permutation in $\mathbb{I}_{0,0}$. This is realized by $suf_{n,l} \times \mathbb{I}_{0,0}$, where $suf_{n,l}$ is the πDD for the set including the n-permutations π in which $\pi_i = i$ holds for $1 \le i \le l$ and the $(n-l)$-suffix is in any order. We can obtain $suf_{n,l}$ by a construction like Algorithm 7, which will be precisely described in Section 3.4.1. Algorithm 4 describes the entire process.

Algorithm 4 Construct $\mathbb{A}$.

```
$\mathbb{I}_{n,l} \leftarrow \pi$DD for $\{e_n\}$
for $i = n-1$ to $0$ do
    for $j = l$ to $0$ do
        if $j < l$ then
            $\mathbb{I}_{i,j} \leftarrow \mathbb{I}_{i+1,j} \cup \mathbb{I}_{i+1,j+1} \cdot \tau_{i+1,j+1}$
        else
            $\mathbb{I}_{i,j} \leftarrow \mathbb{I}_{i+1,j}$
        end if
    end for
end for

$suf_{l,l} \leftarrow \pi$DD for $\{e_n\}$
for $i = l+1$ to $n$ do
    $suf_{i,l} = suf_{i-1,l}$
    for $j = l+1$ to $i-1$ do
        $suf_{i,l} \leftarrow suf_{i,l} \cup suf_{i-1,l} \cdot \tau_{i,j}$
    end for
end for
return $suf_{n,l} \times \mathbb{I}_{0,0}$
```

3.2 Generation of Vincular Pattern-Containing Permutations

Recall that the additional restriction of vincular patterns is adjacency of positions. Therefore, we can generate vincular pattern-containing permutations by a slight modification of step C from Section 3.1.2. We call the modified step C′. $\mathbb{C}'$ denotes the πDD which corresponds to step C′.

If the jth 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 ith position, we must assign $(j+1)$ to $(i+1)$th position. For $\mathbb{C}'$, we define $\mathbb{P}'_{i,j} = \mathbb{P}'_{i-1,j-1} \cdot \tau_{i,j}$ if the jth and the $(j+1)$th elements must be adjacent, and otherwise $\mathbb{P}'_{i,j} = \mathbb{P}'_{i-1,j} \cup \mathbb{P}'_{i-1,j-1} \cdot \tau_{i,j}$ as $\mathbb{P}_{i,j}$.

As shown in Section 3.1.2, $\mathbb{P}'_{i-1,j}$ consists only of permutations π such that $\pi_i \neq j$, and $\mathbb{P}'_{i-1,j-1} \cdot \tau_{i,j}$ consists only of π such that $\pi_i = j$. Thus, if the jth and the $(j+1)$th elements must be adjacent, $\mathbb{P}'_{i,j}$ includes only permutations π such that $\pi_i = j$, and $\mathbb{P}'_{i,j} \cdot \tau_{i+1,j+1}$ includes only permutations π such that $\pi_i = j$ and $\pi_{i+1} = j+1$, that is, j and $j+1$ are adjacent. Therefore, $\mathbb{P}'_{i+1,j+1} = \mathbb{P}'_{i,j+1} \cup \mathbb{P}'_{i,j} \cdot \tau_{i+1,j+1}$ includes only the permutations in which j and $j+1$ are adjacent because $\mathbb{P}'_{i,j} \cdot \tau_{i+1,j+1}$ satisfies the adjacency as above, and $\mathbb{P}'_{i,j+1}$ also satisfies the adjacency recursively. Therefore, we obtain $\mathbb{C}' = \mathbb{P}'_{n,l}$ by Algorithm 5, which is Algorithm 3 with one additional branch.

Algorithm 5 Construct $\mathbb{C}'$.

```
$\mathbb{P}'_{0,0} \leftarrow \pi\mathrm{DD}\ \{e_n\}$
for $i = 1$ to $n$ do
  for $j = 0$ to $l$ do
    if $j > 0$ then
      if $j$th and $(j+1)$th elements must be adjacent then
        $\mathbb{P}'_{i,j} \leftarrow \mathbb{P}'_{i-1,j-1} \cdot \tau_{i,j}$
      else
        $\mathbb{P}'_{i,j} \leftarrow \mathbb{P}'_{i-1,j} \cup \mathbb{P}'_{i-1,j-1} \cdot \tau_{i,j}$
      end if
    else
      $\mathbb{P}'_{i,j} \leftarrow \mathbb{P}'_{i-1,j}$
    end if
  end for
end for
return $\mathbb{P}'_{n,l}$
```

3.3 Generation of Bivincular Pattern-Containing Permutations

Bivincular patterns have the three restrictions: a relative order, adjacencies of positions and consecutiveness of values. Hence, we use step C′ as in Section 3.2 and change step A to A′ in the same way as C was changed to C′. If the jth and the $(j+1)$th values must be consecutive, we define $\mathbb{I}'_{i,j} = \mathbb{I}'_{i+1,j+1} \cdot \tau_{i+1,j+1}$. Algorithm 6 gives the details of this algorithm.

3.4 Generation of Pattern-Avoiding Permutations

In this section, we give an algorithm for generating all σ-avoiding n-permutations. 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

Algorithm 6 Construct $\mathbb{A}'$.

```
$\mathbb{I}_{n,l} \leftarrow \pi$DD for $\{e_n\}$
for $i = n-1$ to 0 do
    for $j = l$ to 0 do
        if $j < l$ then
            if $j$th and $(j+1)$th elements must be consecutive then
                $\mathbb{I}_{i,j} \leftarrow \mathbb{I}_{i+1,j+1} \cdot \tau_{i+1,j+1}$
            else
                $\mathbb{I}_{i,j} \leftarrow \mathbb{I}_{i+1,j} \cup \mathbb{I}_{i+1,j+1} \cdot \tau_{i+1,j+1}$
            end if
        else
            $\mathbb{I}_{i,j} \leftarrow \mathbb{I}_{i+1,j}$
        end if
    end for
end for

$suf_{l,l} \leftarrow \pi$DD for $\{e_n\}$
for $i = l+1$ to $n$ do
    $suf_{i,l} = suf_{i-1,l}$
    for $j = l+1$ to $i-1$ do
        $suf_{i,l} \leftarrow suf_{i,l} \cup suf_{i-1,l} \cdot \tau_{i,j}$
    end for
end for
return $suf_{n,l} \times \mathbb{I}_{0,0}$
```

σ-avoiding n-permutations and S_n denotes the set of all n-permutations. As stated above, $Av_n(\sigma) = S_n \setminus C_n(\sigma)$ holds. The generation of the set of permutations which contain σ was shown above in Sections 3.1 to 3.3 . In general, the time needed 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 now introduce the algorithm for generating S_n.

3.4.1 Generation Algorithm for all n-Permutations

Let $\mathbb{S}_n$ denote the πDD for S_n. We can recursively construct $\mathbb{S}_n$. Suppose we obtained $\mathbb{S}_{n-1}$. We consider $(n-1)$-permutations as n-permutations with $\pi_n = n$. Thus, $\mathbb{S}_{n-1} \cdot \tau_{k,n}$ consists of all n-permutations π such that $\pi_n = k$. Therefore, $\mathbb{S}_n$ can be obtained by computing $\mathbb{S}_{n-1} \cdot \tau_{1,n} \cup \mathbb{S}_{n-1} \cdot \tau_{2,n} \cup \ldots \cup \mathbb{S}_{n-1} \cdot \tau_{n-1,n} \cup \mathbb{S}_{n-1}$. Algorithm 7 realizes this procedure, using loops. Figure 3.3 shows $\mathbb{S}_4$. While the cardinality of S_n is $n!$, the size of $\mathbb{S}_n$ is $O(n^2)$ as shown in Figure 3.3. This demonstrates a high compression ratio of πDDs.

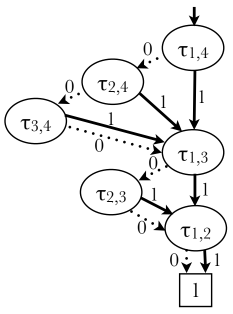


Figure 3.3. The πDD for S_4.

Algorithm 7 Construct $\mathbb{S}_n$.

```
$\mathbb{S}_0 \leftarrow \pi$DD for $\{e_n\}$
**for** $i = 1$ to $n$ **do**
    $\mathbb{S}_i \leftarrow \mathbb{S}_{i-1}$
    **for** $j = 1$ to $i - 1$ **do**
        $\mathbb{S}_i \leftarrow \mathbb{S}_i \cup (\mathbb{S}_{i-1} \cdot \tau_{i,j})$
    **end for**
**end for**
**return** $\mathbb{S}_n$
```

3.4.2 Summary of Generation Algorithm for $Av_n(\sigma)$

Our algorithm for $Av_n(\sigma)$ can be summarized as follows. First, we construct the πDD for 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 $C_n(\sigma)$, and hence obtain $Av_n(\sigma)$. This procedure is illustrated in Figure 3.4.

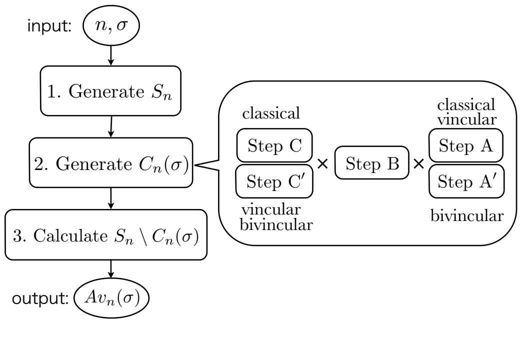


Figure 3.4. The summary of our algorithms for $Av_n(\sigma)$.

3.5 Generation of Permutations with Pattern Occurrence Counts

In this section, we give an algorithm generating $F_n^{(k)}(\sigma)$. Our algorithm is divided into two parts: first we generates $\mathbf{P}_{n,\sigma}$, and then computes $F_n^{(k)}(\sigma) = \{\pi \mid \mathbf{P}_{n,\sigma}(\pi) = k\}$ from $\mathbf{P}_{n,\sigma}$.

3.5.1 Generation Algorithm for $\mathbf{P}_{n,\sigma}$

We gave a generation algorithm for non-weighted occurrence using πDD. Thus, in order to construct the πDD vector for $\mathbf{P}_{n,\sigma}$, we want to extend the generation algorithm for $C_n(\sigma)$ to handle multiplicities as the pattern occurrence counts of σ.

Here, note that some embeddings in the generation of $C_n(\sigma)$ cause duplications. For example, both 413 and 423, which are order isomorphic to 312, are embedded in 4213, and there is no other numerical sequence order isomorphic to 312 which is embedded in 4213. In other words, 312 occurs twice in 4213. The number of such duplications is equal to the pattern occurrence count of a given pattern in a permutation. Since the algorithm described in Section 3.1 uses πDD, such duplications are not counted. But by using the Cartesian product of πDD vectors instead of πDDs, we can generate the multiset of permutations whose multiplicity represents the pattern occurrence count, i.e. $\mathbf{P}_{n,\sigma}$.

3.5.2 Generation Algorithm for $F_n^{(k)}(\sigma)$

We can obtain $\mathbf{P}_{n,\sigma}$ by using Cartesian product of πDD vectors as described in the previous subsection. However, in order to computationally check whether or not strong Wilf-equivalence between σ_1 and σ_2 holds, we must calculate the cardinalities of $F_n^{(k)}(\sigma_1)$ and $F_n^{(k)}(\sigma_2)$ for $0 \le k \le M$, where M denotes the maximum number of occurrences of a given pattern. Note that $M \le \binom{n}{l}$ holds, where l is the length of the pattern, because the number of subsequences of length l in n-permutation is $\binom{n}{l}$. In this subsection, we present an algorithm constructing the πDDs for $F_n^{(k)}(\sigma)$ for $0 \le k \le M$ from the πDD vector for $\mathbf{P}_{n,\sigma}$ which was constructed by the algorithm in the previous subsection.

Let m be $\lfloor \log M \rfloor$. If k is fixed, it is easy to calculate $F_n^{(k)}(\sigma)$ from the πDD vector $\vec{P} = (P_0, P_1, \ldots, P_m)$ as follows:

$$F_n^{(k)}(\sigma) = \bigcap_{0 \le i \le m} \{P_i \mid k_i = 1\} \setminus \bigcup_{0 \le i \le m} \{P_i \mid k_i = 0\},$$

where k_i means the ith bit of the binary representation of k. This algorithm costs $O(m)$ πDD operations. Hence, computation of $F_n^{(k)}(\sigma)$ for $0 \le k \le M$ costs $O(mM)$ πDD operations.

We improve the number of πDD operations from $O(mM)$ to $O(M)$. Let W_k be the πDD for the set of permutations whose multiplicity is k in the given πDD vector $\vec{P}$.

We introduce *don't care* $*$ to the binary representation of integers. Here, $W_{(*\ldots*0k_i\ldots k_0)_2} \cup W_{(*\ldots*1k_i\ldots k_0)_2} = W_{(*\ldots**k_i\ldots k_0)_2}$ and $W_{(*\ldots*0k_i\ldots k_0)_2} \cap W_{(*\ldots*1k_i\ldots k_0)_2} = \emptyset$ hold. Hence,

$$\begin{aligned} W_{(*\ldots*1k_i\ldots k_0)_2} &= W_{(*\ldots**k_i\ldots k_0)_2} \cap P_{i+1}, \\ W_{(*\ldots*0k_i\ldots k_0)_2} &= W_{(*\ldots**k_i\ldots k_0)_2} \setminus W_{(*\ldots*1k_i\ldots k_0)_2}. \end{aligned}$$

Therefore, we can calculate W_k for $0 \le k \le M$ from $W_{(*\ldots*)_2} = S_n$ by repeating calculations of the above recursions. The number of valid binary representations whose prefix can be consecutive "don't care" symbols is $2^0 + 2^1 + \ldots + 2^m = 2^{m+1} - 1 = O(M)$. For each calculation of W_k, we use only one πDD operation. Therefore, we can generate $F_n^{(k)}(\sigma)$ for $0 \le k \le M$ by $O(M)$ πDD operations based on the recursions. Algorithm 8 gives a pseudo code of this algorithm. This algorithm temporarily uses $W_{(0\ldots0k_i\ldots k_0)_2}$ to represent $W_{(*\ldots*k_i\ldots k_0)_2}$.

Algorithm 8 Generate W_k for all $0 \le k \le M$ from $\vec{P} = (P_0, P_1, \ldots, P_m)$.

```
$W_0 \leftarrow \pi$DD for $S_n$
for $i = 0$ to $m$ do
  for $bin = 2^i$ to $2^{i+1} - 1$ do
    $W_{bin} \leftarrow W_{bin-2^i} \cap P_i$
    $W_{bin-2^i} \leftarrow W_{bin-2^i} \setminus W_{bin}$
  end for
end for
```

3.5.3 Summary of Generation Algorithm for $F_n^{(k)}(\sigma)$

Our algorithm can be summarized as follows. First, we construct the πDDs for $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$ by the algorithms described in Section 3.1. Second, we transform these πDDs into the πDD vectors, that is, $\vec{P} = (\mathbb{C})$ and $\vec{Q} = (\mathbb{B} \times \mathbb{A})$. Third, we calculate the Cartesian product $\vec{P} \times \vec{Q}$, which is the πDD vector for $\mathbf{P}_{n,\sigma}$. Finally, we calculate the πDDs for $F_n^{(k)}(\sigma)$ from the πDD vector for $\mathbf{P}_{n,\sigma}$ using Algorithm 8. This process is illustrated in Figure 3.5.

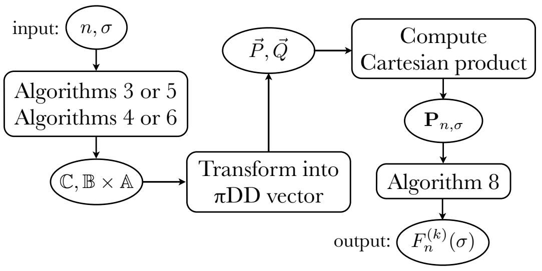

input: n, σ
Algorithms 3 or 5
Algorithms 4 or 6
$\mathbb{C}, \mathbb{B} \times \mathbb{A}$
Transform into πDD vector
$\vec{P}, \vec{Q}$
Compute Cartesian product
$\mathbf{P}_{n,\sigma}$
Algorithm 8
output: $F_n^{(k)}(\sigma)$

Figure 3.5. The summary of our algorithm for $F_n^{(k)}(\sigma)$.

Chapter 4

Experimental Results

We implemented our algorithm in C++ and carried out computational experiments on a 3.20 GHz Intel Core i7-3930K CPU machine with Ubuntu 12.04 LTS 64-bit OS and 64 GB memory. We compared the performance of our algorithm to that of the naive method. The naive method generates all n-permutations and, for each n-permutation, enumerates pattern occurrence count of σ by checking the order isomorphism between all k-subsequences and σ.

Table 4.1. Computation time (sec) for generating classical pattern-avoiding permutations.

n		πDD Method				Naive Method			
		l				l			
		2	3	4	5	2	3	4	5
8	best	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	average	0.000	0.002	0.004	0.001	0.008	0.017	0.053	0.007
	worst	0.000	0.008	0.012	0.008	0.016	0.020	0.064	0.080
9	best	0.000	0.000	0.004	0.004	0.052	0.116	0.468	0.880
	average	0.004	0.005	0.009	0.009	0.056	0.119	0.499	0.926
	worst	0.008	0.016	0.016	0.024	0.060	0.124	0.540	1.004
10	best	0.004	0.004	0.016	0.024	0.472	1.140	6.000	17.089
	average	0.008	0.011	0.028	0.036	0.478	1.173	6.453	18.223
	worst	0.012	0.020	0.044	0.060	0.484	1.200	6.932	19.697
11	best	0.000	0.012	0.044	0.092	5.240	13.053	89.494	417.958
	average	0.008	0.029	0.101	0.174	5.290	13.216	95.727	435.211
	worst	0.016	0.052	0.152	0.276	5.340	13.365	101.878	453.604
12	best	0.004	0.024	0.152	0.428	—	—	—	—
	average	0.014	0.087	0.444	0.921	—	—	—	—
	worst	0.024	0.156	0.780	1.392	—	—	—	—-
13	best	0.016	0.048	0.568	1.824	—	—	—	—
	average	0.022	0.284	1.787	4.309	—	—	—	—
	worst	0.022	0.544	3.072	6.888	—	—	—	—
14	best	0.008	0.116	1.960	6.448	—	—	—	—
	average	0.032	1.029	6.769	19.036	—	—	—	—
	worst	0.056	1.968	12.021	32.370	—	—	—	—
15	best	0.016	0.300	5.688	23.814	—	—	—	—
	average	0.052	3.513	24.771	85.655	—	—	—	—
	worst	0.088	6.860	48.415	160.562	—	—	—	—

Table 4.2. Memory consumption (kB) for generating classical pattern-avoiding permutations.

n		$\lvert Av_n(\sigma)\rvert$				πDD Method				Naive Method			
		l				l				l			
		2	3	4	5	2	3	4	5	2	3	4	5
8	best	1	1430	15485	33184	1600	1596	1596	1596	1068	1064	1064	1064
	average	1	1430	15656	33277	1600	1598	1916	1599	1068	1067	1067	1067
	worst	1	1430	15793	33325	1600	1600	1972	1600	1068	1068	1072	1072
9	best	1	4862	91245	258757	1600	1964	2748	2744	1072	1068	1068	1068
	average	1	4862	93096	260742	1600	2233	2933	2768	1072	1071	1071	1071
	worst	1	4862	94776	261863	1600	2768	4208	2792	1072	1072	1076	1076
10	best	1	16796	555662	2136978	1600	2760	4212	7156	1068	1068	1068	1068
	average	1	16796	574150	2171460	1784	3509	6856	7376	1070	1071	1071	1071
	worst	1	16796	591950	2192390	1968	4260	7444	7700	1072	1072	1076	1076
11	best	1	58786	3475090	18478134	1600	4208	7676	13720	1072	1072	1068	1068
	average	1	58786	3648275	19011623	2186	5876	17641	25233	1072	1072	1071	1071
	worst	1	58786	3824112	19358590	2772	6792	25084	27000	1072	1072	1072	1076
12	best	1	208012	22214707	165857600	1976	7112	25316	49108	—	—	—	—
	average	1	208012	23771768	173553425	3112	16106	48405	93525	—	—	—	—
	worst	1	208012	25431452	178904675	4248	25084	94788	102940	—	—	—	—-
13	best	1	742900	144640291	1535336290	2764	12708	51436	188416	—	—	—	—
	average	1	742900	158260498	1641499314	4954	32130	168848	337123	—	—	—	—
	worst	1	742900	173453058	1720317763	7144	51084	201264	408460	—	—	—	—
14	best	1	2674440	956560748	14584260700	2760	24440	187372	396432	—	—	—	—
	average	1	2674440	1073474327	16006197603	7754	111634	542621	1282547	—	—	—	—
	worst	1	2674440	1209639642	17132629082	12748	190460	779640	1587600	—	—	—	—
15	best	1	9694845	6411521056	141603589300	4228	47620	389140	1543108	—	—	—	—
	average	1	9694845	7401901167	160274747099	9112	234836	1560720	4986352	—	—	—	—
	worst	1	9694845	8604450011	176055309619	13996	406440	3092572	6471388	—	—	—	—

4.1 Results for Classical Pattern-Avoiding Permutations

Tables 4.1 and 4.2 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 $l = 2, 3, 4$, and 5. Note that computation time of our πDD method is time to construct $Av_n(\sigma)$, and computation time of the naive method is time to output all pattern-avoiding permutations to /dev/null. Since output is not required for our purpose as described in Section 1.3, computation time of πDD method does not include output time.

For computation time, in almost all cases, our algorithm is faster than the naive method. For example, in $n = 11$, our method requires only 0.3% of the time required by the naive one. It should be noted that there are differences between the best and worst performance for the same case in the results of our algorithm, while the naive method hardly shows any differences. However, in almost all worst-case scenarios, the performance of our algorithm is better than the best-case scenario of the naive one. Computation time of both methods increase exponentially with respect to n, but our algorithm has a smaller growth rate than the naive method.

Figures 4.1 and 4.2 also show the results for classical patterns. These results show the average time and memory consumption for all patterns, where n is fixed at 10 and l varies. Estimate represents the estimated memory consumption for storing all pattern-avoiding permutation on memory, where we suppose that the representation of one number costs one byte, i.e. $n \cdot |Av_n(\sigma)|$. Naive method hardly uses memory, because it stores no information. Memory consumption of the πDD method is greater than naive method, but it is very smaller than the estimate especially when l is near n. This shows that πDD can achieve high compression when the cardinality of the set is near $n!$, such as in the case of S_n. The results also show the computation time depends on the size of πDDs. In contrast,

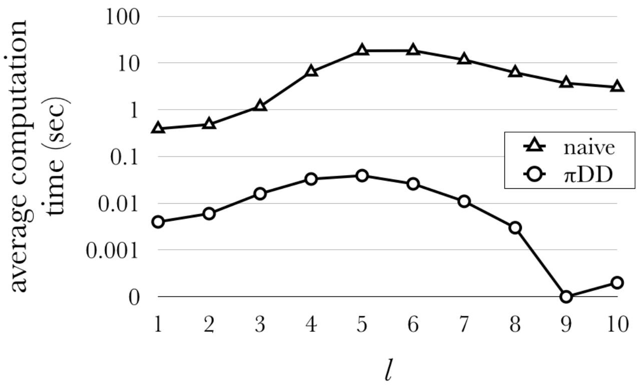


Figure 4.1. Computation time when $n = 10$.

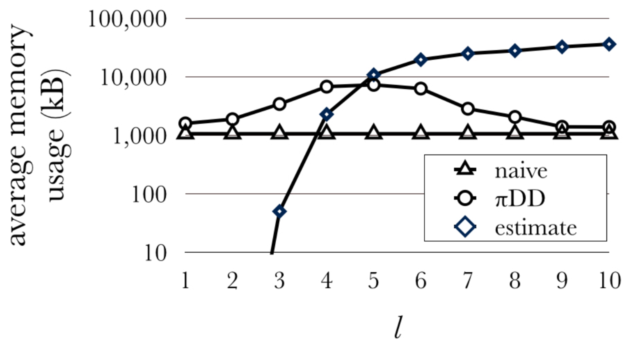


Figure 4.2. Memory usage when $n = 10$.

the computation time of the naive method is in proportion to $\binom{n}{l}$, which is the number of subsequences that must be checked.

4.2 Results for Vincular and Bivincular Patterns

Table 4.3 presents the results for vincular patterns. We only generated Baxter permutations, because there are a huge number of vincular patterns for each l, namely $l! \cdot 2^{l-1}$. Baxter permutations are defined as avoiding the two vincular patterns, $3\underline{14}2$ and $2\underline{41}3$, and appear in many mathematical problems [1, 8, 10]. Our algorithm generates $Av_n(3\underline{14}2, 2\underline{41}3) = S_n \setminus (C_n(3\underline{14}2) \cup C_n(2\underline{41}3))$. On the other hand, the naive method checks the order isomorphism between all l-subsequences and the two patterns simultaneously. $B(n)$ denotes the number of Baxter permutations of length n.

The performance of our algorithm for vincular patterns is as good as that for classical patterns. Our algorithm is faster than the naive method.

Table 4.3. Experimental results for generating Baxter permutations.

n	$B(n)$	πDD Method		Naive Method	
		Time (s)	Memory (kB)	Time (s)	Memory (kB)
8	10754	0.016	2760	0.052	3124
9	58202	0.028	4164	0.448	3128
10	326240	0.060	12984	5.792	3124
11	1882960	0.212	26828	79.405	3128
12	11140560	0.908	98924	1147.583	3124
13	67329992	3.678	383272	—	—
14	414499438	13.419	824732	—	—
15	2593341586	50.499	3151164	—	—
16	16458756586	193.704	12403488	—	—
17	105791986682	745.779	40788188	—	—

Table 4.4. the πDD for Baxter permutations ($n = 15$).

	#permutations	#nodes in πDD
S_n	1307674368000	105
$C_n(3\underline{14}2) \cup C_n(2\underline{41}3)$	1305081026414	4094585
$Av_n(3\underline{14}2, 2\underline{41}3)$	2593341586	2158472

When $n = 15$, the time for calculating the difference $S_n \setminus C_n(\sigma)$ is 1.110 seconds, which is about 2% of the entire computation time. In general, calculating the set difference for sets with large cardinality without πDDs is not efficient, but, πDD's difference operation is not a bottleneck in this problem. Most of the computation time is due to the Cartesian product operations between three πDDs, which require 46.020 seconds.

The number of permutations and the size of the corresponding πDD are shown in Table 4.4, where it is clear that πDD achieves a high compression ratio.

We show the results for bivincular patterns. Specifically, we generated $Av_n\begin{pmatrix}\overline{12}3\\ \underline{23}1\end{pmatrix}$, which is known to be related with chord diagrams, (2+2)-free posets and ascent sequences [7]. Table 4.5 presents the result of generating $Av_n\begin{pmatrix}\overline{12}3\\ \underline{23}1\end{pmatrix}$. Both our algorithm and the naive method show better performance than the result on Baxter permutations because the pattern length, number of patterns, and the cardinality of $Av_n\begin{pmatrix}\overline{12}3\\ \underline{23}1\end{pmatrix}$ are all decreasing. For $n = 13$, the πDD method can generate the permutations in less than 1 second while the naive one requires about 2 hours.

Table 4.5. Experimental results for bivincular pattern.

n	$Av_n\begin{pmatrix}\overline{12}3\\ \underline{23}1\end{pmatrix}$	πDD Method		Naive Method	
		Time (s)	Memory (kB)	Time (s)	Memory (kB)
8	5335	0.004	3084	0.028	3128
9	31240	0.004	3084	0.208	3128
10	201608	0.020	4196	2.428	3128
11	1422074	0.052	12948	32.298	3128
12	10886503	0.184	26436	474.173	3128
13	89903100	0.916	98656	7458.270	3128
14	796713190	3.808	382712	—	—
15	7541889195	15.357	1510640	—	—
16	75955177642	62.408	3327380	—	—
17	810925547354	254.896	12887884	—	—
18	9148832109645	1016.132	50354060	—	—

4.3 Results for Permutations with Pattern Occurrence Counts

Table 4.6 shows computation time for the entire process of generating $F_n^{(k)}(\sigma)$. The table represents the best, the worst, and the average computation time over all patterns with length $l = 2,3,4$, and 5, respectively. Note that computation time for our πDD method only include the constructions of πDDs, meaning that we do not decompress πDDs and thus do not output permutations explicitly. The naive method only counts $|F_n^{(k)}(\sigma)|$, and not explicitly outputs permutations too. For $n = 10$ and $l = 5$, our algorithm is about 20 times faster than the naive method. Our algorithm takes the maximum computation time when l is near $n/3$, while the naive method takes the maximum time when l is near $n/2$. We consider the reasons for the difference as follows. At first, the computation time of πDD depends on the size of πDD, while that of the naive method depends on the number of subsequences $\binom{n}{l}$. In practical cases, the size of a πDD tends to become small when the set of permutations is sparse or very dense. The longer the length of a pattern is, the fewer the pattern occurrence counts of the pattern in each permutation is. This means that $F_n^{(k)}(\sigma)$ tends to be dense for small k and to be sparse for large k when σ is a long pattern. We imagine this is a reason why l on the peak of computation time of our algorithm is smaller than l on the peak of the naive method.

Table 4.7 shows memory consumption and the size of πDDs for generating $F_n^{(k)}(\sigma)$. We do not describe the memory consumption of the naive method because the naive method does not store the permutations, and uses a small memory. Tables 4.6 and 4.7 suggest the computation time of πDDs is proportional to the size of πDDs. Memory consumption of our algorithm grows exponentially with respect to n. We cannot calculate $F_{13}^{(k)}(\sigma)$ because of insufficient memory. However the total size of πDDs for $F_n^{(k)}(\sigma)$ is smaller than the number of all n-permutations $n!$. Moreover, since some πDDs share their equivalent subgraphs, the actual number of the total nodes of the πDDs on memory may be smaller than the results in Table 4.7. This shows that πDDs achieve compact representations of $F_n^{(k)}(\sigma)$.

Table 4.6. Computation time (sec) for generating $F_n^{(k)}(\sigma)$.

n		πDD Method				Naive Method			
		l				l			
		2	3	4	5	2	3	4	5
8	best	0.032	0.044	0.032	0.016	0.036	0.052	0.060	0.044
	average	0.036	0.052	0.043	0.021	0.038	0.057	0.071	0.054
	worst	0.040	0.064	0.056	0.028	0.040	0.064	0.080	0.072
9	best	0.076	0.212	0.232	0.068	0.316	0.676	1.012	0.968
	average	0.078	0.295	0.274	0.087	0.322	0.714	1.088	1.037
	worst	0.080	0.356	0.308	0.120	0.328	0.740	1.164	1.128
10	best	0.276	1.912	2.820	0.956	3.792	10.301	20.141	25.522
	average	0.310	2.604	3.390	1.232	3.900	10.891	21.489	27.314
	worst	0.344	3.288	3.908	1.524	4.008	11.393	23.013	29.482
11	best	1.436	14.165	30.190	11.673	51.595	169.331	467.185	750.307
	average	1.642	18.943	38.814	15.683	53.673	179.724	488.088	774.683
	worst	1.848	24.098	48.331	19.809	55.752	188.248	509.804	806.186
12	best	6.316	96.114	365.275	144.141	—	—	—	—
	average	7.064	136.343	494.926	231.141	—	—	—	—
	worst	7.812	180.711	655.829	301.135	—	—	—	—

Table 4.7. Memory consumption and the size of πDDs for generating $F_n^{(k)}(\sigma)$.

n		memory consumption (kB)				$\sum_{k=0}^{M}\{$the size of πDDs for $F_n^{(k)}(\sigma)\}$			
		l				l			
		2	3	4	5	2	3	4	5
8	best	37548	37812	37808	37236				
	average	37548	38075	37819	37280	10535	20941	11501	4144
	worst	37548	38340	38024	37288				
9	best	38340	41528	41392	38604				
	average	38736	42542	42372	39076	42496	139315	86321	30843
	worst	39132	43156	42940	39660				
10	best	42480	152316	268348	78532				
	average	43272	212905	275364	141172	167368	962945	726510	262833
	worst	44064	275604	279712	154728				
11	best	152328	1032672	2064776	1031504				
	average	152484	1092093	2142647	1067605	658823	6684948	6807355	2548799
	worst	152640	1160728	2224848	1128760				
12	best	520216	4209304	16461940	8263752				
	average	527320	6863648	18459651	9723839	2585682	45156225	69564400	27639470
	worst	534424	8232056	32355008	16375948				

Table 4.8 provides candidates of strongly Wilf-equivalent classes. Two patterns σ_1 and σ_2 are placed in the same cell of the table if $|F_n^{(k)}(\sigma_1)| = |F_n^{(k)}(\sigma_2)|$ was confirmed for $1 \leq n \leq 12$ and non-negative integers k. It is trivial that a pattern π and its reverse π^r are always strongly Wilf-equivalent. Note that a pattern π and its inverse π^{-1} are not necessarily strong Wilf-equivalent, while these are always Wilf-equivalent. Table 4.8 shows many non-trivial and unproved candidates of strongly Wilf-equivalent classes.

Table 4.8. Candidates of strongly Wilf-equivalent classes according to the experimental results.

l	candidates of strongly Wilf-equivalent classes	
2	12,21	
3	123,321	132,213,231,312
4	1234,4321	1243,2134,3421,4312
	1324,4231	1342,1423,2314,2431,3124,3241,4132,4213
	1432,2341,3214,4123	2143,3412
	2413,3142	
5	12345,54321	12354,21345,45321,54312
	12435,13245,53421,54231	12453,12534,23145,31245,35421,43521,54132,54213
	12543,32145,34521,54123	13524,14253,24135,31425,35241,42531,52413,53142
	13425,14235,52431,53241	13452,15234,23415,25431,41235,43251,51432,53214
	13254,21435,45231,53412	13542,15243,24531,32415,34251,42135,51423,53124
	14325,52341	14352,15324,24315,25341,41325,42351,51342,52314
	14523,32541,34125,52143	14532,15423,23541,32451,34215,43125,51243,52134
	15342,24351,42315,51324	15432,23451,43215,51234
	21354,45312	21453,21534,23154,31254,35412,43512,45132,45213
	21543,32154,34512,45123	23514,25134,25413,31452,35214,41253,41532,43152
	24153,31524,35142,42513	24513,25143,31542,32514,34152,35124,41523,42153
	25314,41352	

Chapter 5

Conclusion

In this thesis, we introduced algorithms for generating several pattern-avoiding permutations using πDDs. Experimental results show that our algorithms are faster than the naive method and cost less memory than the naive storing. In addition, our approaches output πDDs on memory, which has rich operations. This means that we can submit additional queries such as membership test or random sampling for the set of pattern-avoiding permutations efficiently and immediately.

Furthermore, we gave an algorithm implicitly generating $\mathbf{P}_{n,\sigma}$ for given n and σ using πDD vectors, and an algorithm computing the πDDs representing $F_n^{(k)}(\sigma)$ for each k from the πDD vector for $\mathbf{P}_{n,\sigma}$. These algorithms was practically faster than a naive method, and provided some candidates of strongly Wilf-equivalent classes.

Future work is to improve further the computation time and memory consumption of our algorithms, and to compare our algorithms and other algorithms for some particular patterns, for example Baxter permutations [4]. 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 πDDs as floorplan databases. For the algorithm generating $F_n^{(k)}(\sigma)$, experiments for larger n and k, multiple patterns, and other general patterns are also interesting for us. We wish that someone will prove some strong Wilf-equivalences which are described in this paper as candidates.

Acknowledgements

I am indebted to my supervisor, Prof. Shin-ichi Minato, for his advise and help, which improved both my research life and this thesis. I am grateful to Prof. Zeugmann Thomas and Prof. Hiroki Arimura, who help me to suggest research ideas and improve my writing skill. I am also indebted Dr. Takahisa Toda, Dr. Charles Harold Jordan, and all members of JST ERATO Minato Project and Laboratory for Algorithmics, who help me with my research and this thesis. Finally, I thank my family for their love and support.

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

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

[5] 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.

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

[7] 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, Oct 2010.

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

[9] 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.

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

[11] Anisse Kasraoui. New Wilf-equivalence results for vincular patterns. *European Journal of Combinatorics*, 34(2):322–337, 2013.

[12] 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.

[13] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[14] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[15] Darko Marinov and Radoš Radoičić. Counting 1324-avoiding permutations. *The Electronic Journal of Combinatorics*, 9(2), 2003.

[16] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. *Proc. of 30th ACM/IEEE Design Automation Conf. (DAC 1993)*, pages 272–277, 1993.

[17] Shin-ichi Minato. πDDs: A new decision diagram for efficient problem solving in permutation space. *Proc. of 14th International Conference on Theory and Applications of Satisfiability Testing (SAT 2011)*, pages 90–104, 2011.

[18] Vaughan R. Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. *Proc. of the Fifth Annual ACM Symposium on Theory of Computing (STOC 1973)*, pages 268–277, 1973.

[19] Nathan Reading. Generic rectangulations. *European Journal of Combinatorics*, 33(4):610–623, 2012.

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

[21] Einar Steingrímsson. Some open problems on permutation patterns. *London Mathematical Society Lecture Note Series*, pages 239–263, 2013.

[22] Julian West. Sorting twice through a stack. *Theoretical Computer Science*, 117:303–313, 1993.

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

[24] Bo Yao, Hongyu Chen, Chung-Kuan Cheng, and Ronald Graham. Floorplan representations: Complexity and connections. *ACM Trans. Des. Autom. Electron. Syst.*, 8(1):55–80, 2003.