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
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

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Related Work</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Thesis Structure</td>
<td>6</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Permutations</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Permutation Patterns</td>
<td>8</td>
</tr>
<tr>
<td>2.3 $\pi$DDs</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 ZDDs</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 $\pi$DDs</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Multiset of Permutations</td>
<td>12</td>
</tr>
<tr>
<td>2.5 $\pi$DD Vectors</td>
<td>12</td>
</tr>
<tr>
<td>3 Main Results</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Generation of Classical Pattern-Containing Permutations</td>
<td>16</td>
</tr>
<tr>
<td>3.1.1 Construction of $B$</td>
<td>18</td>
</tr>
<tr>
<td>3.1.2 Construction of $C$</td>
<td>18</td>
</tr>
<tr>
<td>3.1.3 Construction of $A$</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Generation of Vincular Pattern-Containing Permutations</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Generation of Bivincular Pattern-Containing Permutations</td>
<td>21</td>
</tr>
<tr>
<td>3.4 Generation of Pattern-Avoiding Permutations</td>
<td>21</td>
</tr>
<tr>
<td>3.4.1 Generation Algorithm for all $n$-Permutations</td>
<td>22</td>
</tr>
<tr>
<td>3.4.2 Summary of Generation Algorithm for $\text{Av}_n(\sigma)$</td>
<td>23</td>
</tr>
<tr>
<td>3.5 Generation of Permutations with Pattern Occurrence Counts</td>
<td>24</td>
</tr>
<tr>
<td>3.5.1 Generation Algorithm for $P_{n,\sigma}$</td>
<td>24</td>
</tr>
<tr>
<td>3.5.2 Generation Algorithm for $F_n^{(k)}(\sigma)$</td>
<td>24</td>
</tr>
<tr>
<td>3.5.3 Summary of Generation Algorithm for $F_n^{(k)}(\sigma)$</td>
<td>25</td>
</tr>
</tbody>
</table>
4 Experimental Results

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 (PiDDs, or πDDs), 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 $\sigma$ occurs in a permutation $\pi$ if there is a subsequence in $\pi$ which is order isomorphic to $\sigma$. Two numerical sequences $a = a_1a_2\ldots a_n$ and $b = b_1b_2\ldots b_m$ 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, $\pi$ avoids $\sigma$ if $\sigma$ does not occur in $\pi$. Permutations that avoid a pattern $\sigma$ are called $\sigma$-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 $\sigma_1$ and $\sigma_2$ are Wilf-equivalent if the number of $n$-permutations which avoid $\sigma_1$ is same as the number of $n$-permutations for $\sigma_2$ for all positive integers $n$. In [20], the nontrivial Wilf-equivalence between 4132-avoiding and 3142-avoiding was discovered. More generally, two patterns $\sigma_1$ and $\sigma_2$ are strongly Wilf-equivalent [11] if $|F_n^{(k)}(\sigma_1)| =
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 $\sigma$ 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. Bijectons 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 potentially strongly Wilf-equivalent.

Our algorithms are based on Permutation Decision Diagrams (PiDDs, or $\pi$DDs), a data structure for compactly representing 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 $\pi$DDs and not on the number of permutations they represent. If the $\pi$DD 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. $12\ldots n$), 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 \( \pi \)DDs. 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 $\{1, 2, \ldots, n\}$ 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 $\pi$ denotes the first $k$ numbers in $\pi$, and the $k$-suffix of $\pi$ denotes the last $k$ numbers in $\pi$. For example, $\pi = 4312$ is a 4-permutation, $\pi_3 = 1$, and the 2-prefix of $\pi$ is 43. An element $\pi_i$ in $\pi$ 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 $1 \leq i \leq n$.

\[
\begin{align*}
45213 \cdot 41352 &= 52143 \\
\begin{array}{c}
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
\end{array} & \begin{array}{c}
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
\end{array} & \begin{array}{c}
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
\end{array} & \begin{array}{c}
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
\end{array} & \begin{array}{c}
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
\end{array}
\end{align*}
\]

Figure 2.1. An example of multiplication of permutations.
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 $k$th 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 $\pi$.

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 $k$th 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 $54213 = 31245 \cdot \tau_{1,2} \cdot \tau_{2,3} \cdot \tau_{1,4} \cdot \tau_{3,5}$. 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 $\pi$ contains a permutation $\sigma$, or $\sigma$ occurs in $\pi$ if there is at least one subsequence in $\pi$ which is order isomorphic to $\sigma$, where the subsequence need not consist of consecutive numbers in $\pi$. In other words, let $l$ be the length of $\sigma$. Then, $\pi$ contains $\sigma$ if there are indexes $1 \leq i_1 < i_2 < \ldots < i_l \leq n$ such that $\pi_{i_k} < \pi_{i_{k+1}}$ if and only if $\sigma_{i_k} < \sigma_{i_{k+1}}$, for all pairs of $x$ and $y$. Such $\sigma$ is called a pattern. For example, the permutation 4213 contains the pattern 312 because 423 and 413 are order isomorphic to the pattern. Conversely, $\pi$ avoids $\sigma$ if $\pi$ does not contain $\sigma$.

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 $i$th 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 $312$. Both 423 and 413 are order isomorphic to 312, but 423 does not match $312$ 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 312.

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 $\overline{123}_3 312$ 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 $\overline{123}_3 312$. 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 $\sigma$ in $\pi$ is the number of distinct subsequences in the permutation $\pi$ which are order isomorphic to the pattern $\sigma$. 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 $\sigma$ occurs exactly $k$ times. Two patterns $\sigma_1$ and $\sigma_2$ 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 $\sigma_1$ and $\sigma_2$ 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 $\sigma$, 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
A 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.
2.3.2 \( \pi \text{DDs} \)

We introduce \( \pi \text{DDs} \), 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 \( \pi \text{DDs} \). We introduce the order of transpositions \( < \) so that \( \pi_{x_1,y_1} < \pi_{x_2,y_2} \) 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 \( \pi \text{DD} \). In \( \pi \text{DDs} \), 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 \( \pi \text{DDs} \) 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 \( \pi \text{DD} \).

Table 2.1 shows the \( \pi \text{DD} \) operations used in this thesis. Here the Cartesian product of \( \pi \text{DDs} \) 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 \( \pi \text{DDs} \). 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. \( \pi \text{DD} \) operations on two \( \pi \text{DDs} \) \( P \) and \( Q \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P : \text{Top} )</td>
<td>return the transposition ( \pi_{x,y} ) by which the root node is labeled.</td>
</tr>
<tr>
<td>( P \cap Q )</td>
<td>return intersection ( { \pi \mid \pi \in P \text{ and } \pi \in Q } ).</td>
</tr>
<tr>
<td>( P \cup Q )</td>
<td>return union ( { \pi \mid \pi \in P \text{ or } \pi \in Q } ).</td>
</tr>
<tr>
<td>( P \setminus Q )</td>
<td>return difference ( { \pi \mid \pi \in P \text{ and } \pi \notin Q } ).</td>
</tr>
<tr>
<td>( P \cdot \pi_{x,y} )</td>
<td>return swap ( { \pi \cdot \pi_{x,y} \mid \pi \in P } ).</td>
</tr>
<tr>
<td>( P \times Q )</td>
<td>return Cartesian product ( { \alpha \cdot \beta \mid \alpha \in P \text{ and } \beta \in Q } ).</td>
</tr>
</tbody>
</table>
2.4 Multiset of Permutations

In this section, we introduce some definitions and notations for multisets of permutations. Let \( P = \langle P, f \rangle \) 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 \( \pi \) and we use the notation \( P(\pi) \) instead of \( f(\pi) \) for convenience. We define \( P \uplus Q \) as the multiset sum of two multisets \( P \) and \( Q \), where \( (P \uplus Q)(\pi) = P(\pi) + Q(\pi) \) holds for all permutations \( \pi \). Scalar multiplication of an integer \( k \) and a multiset \( P \) is defined by \( k \cdot P = \langle P, k \cdot f \rangle \). Cartesian product of two multisets \( P = \langle P, f \rangle \) and \( Q = \langle Q, g \rangle \) is defined by \( P \times Q = \langle P \times Q, h(\pi, \pi') = f(\pi) \cdot g(\pi') \rangle \), where \( \pi \in P \), \( \pi' \in Q \), and \( P \times Q \) means Cartesian product of permutation sets like that of \( \pi \) DDs.

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

2.5 \( \pi \) DD Vectors

In this thesis, we want to handle multisets of permutations. But \( \pi \) DDs cannot represent multisets. To overcome this problem, we use \( \pi \) DD vectors, which were proposed in [12]. A \( \pi \) DD vector is a data structure for a multiset of permutations based on a binary representation using multiple \( \pi \) DDs. A \( \pi \) DD vector consists of an array of \( \pi \) DDs. Let \( M \) be the maximum multiplicity of a permutation in a given multiset. We denote a \( \pi \) DD vector by \( \vec{P} = (P_0, P_1, \ldots, P_m) \), where each \( P_i \) is a \( \pi \) DD and \( m = \lceil \log M \rceil \). Let \( \vec{P}(\pi) \) be the multiplicity of \( \pi \) in \( \vec{P} \). If a permutation \( \pi \) is in \( P_i \), the \( i \)th 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 \( \pi \) DD vector. In actuality, \( \pi \) DDs in a \( \pi \) DD vector share common subgraphs as illustrated in [12]. Thus, more efficient compression can be expected.

\( \pi \) DD vectors have multiset operations. Table 2.2 shows some \( \pi \) DD vector operations which were used in this thesis. Most operations in Table 2.2 were proposed in [12].

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vec{P} \uplus \vec{Q} )</td>
<td>Return multiset sum ( \vec{R} ) such that ( \vec{R}(\pi) = \vec{P}(\pi) + \vec{Q}(\pi) ) holds.</td>
</tr>
<tr>
<td>( \vec{P} \cdot \tau_{xy} )</td>
<td>Return swap ( \vec{P}' = (P_0 \cdot \tau_{xy}, P_1 \cdot \tau_{xy}, \ldots, P_m \cdot \tau_{xy}) ).</td>
</tr>
<tr>
<td>( \vec{P}.\text{numberof}(\pi) )</td>
<td>Return ( \vec{P}(\pi) ).</td>
</tr>
<tr>
<td>( \vec{P} \times \vec{Q} )</td>
<td>Return Cartesian product ( \vec{R} ) such that ( \vec{R}(\alpha \cdot \beta) = \vec{P}(\alpha) \cdot \vec{Q}(\beta) ).</td>
</tr>
</tbody>
</table>
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:

\[
\overrightarrow{P} \times \overrightarrow{Q} = (\overrightarrow{P} \times \overrightarrow{Q}^L) \cup ((\overrightarrow{P} \times \overrightarrow{Q}^R) \cdot \tau_{x,y}),
\]

where multiset sum \( \cup \) 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 \( \overrightarrow{Q} \) can have distinct transpositions in their root nodes. We need to define \( \overrightarrow{Q}^L \) and \( \overrightarrow{Q}^R \), and \( \tau_{x,y} \) appropriately.

We use \( \tau_{x,y} \), which is the smallest transposition in the root nodes of \( Q_1, Q_2, \ldots, Q_m \), where the order of transpositions is introduced in Section 2.3.2. For example, the πDD vector illustrated in Figure 2.6, \( \tau_{x,y} \) 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 \( \overrightarrow{Q}^L \) and \( \overrightarrow{Q}^R \) as follows:

\[
\overrightarrow{Q}^L = (Q_0^L, Q_1^L, \ldots, Q_m^L), \text{ where } Q_i^L = \begin{cases} Q_i & \text{if } Q_i.\text{Top} \neq \tau_{x,y}, \\ Q_i^L & \text{otherwise.} \end{cases}
\]

\[
\overrightarrow{Q}^R = (Q_0^R, Q_1^R, \ldots, Q_m^R), \text{ where } Q_i^R = \begin{cases} \emptyset & \text{if } Q_i.\text{Top} \neq \tau_{x,y}, \\ Q_i^R & \text{otherwise.} \end{cases}
\]

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, $\bar{Q}(e_n) \geq 0$ and $\bar{Q}(\pi) = 0$ for the other permutations $\pi$ hold. Then, $\vec{P} \times \vec{Q}$ equals the scalar multiplication $\bar{Q}(e_n) \cdot \vec{P}$. Here, we define $(k_1 k_{l-1} \ldots k_0)_2$ 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 $\pi$DD vector based on the multiset sum of $2^i \cdot \vec{P} = (\emptyset, \ldots, \emptyset, P_0, \ldots, P_m)$. In conclusion, Algorithm 2 shows how to compute the Cartesian product of $\pi$DD vectors.

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

```
πDD vector $\vec{R} \leftarrow (\emptyset)$
for $i = 0$ to $l$ do
    if $k_i = 1$ then
        $\vec{R} \leftarrow \vec{R} \oplus \vec{P}$
    end if
    $\vec{P} \leftarrow (\emptyset, P_0, P_1, \ldots, P_{m+i})$
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 $t_{0,0}$ and the 1-terminal node is labeled by $t_{1,1}$ for convenience.}

transposition $t \leftarrow t_{0,0}$

for $i = 0$ to $m$ do
    if $Q_i\text{.Top} < t$ then
        $t \leftarrow Q_i\text{.Top}$
    end if
end for

{Branch based on $t$.}

if $t = t_{0,0}$ then
    {Every $Q_i$ consists only of the 0-terminal node.}
    return ($\emptyset$)
else if $t = t_{1,1}$ 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 $Q_i\text{.Top} = t$ then
            $Q^L_i \leftarrow Q', Q^R_i \leftarrow Q'$
        else
            $Q^L_i \leftarrow Q_i, Q^R_i \leftarrow \emptyset$
        end if
    end for
    return $(\vec{P} \times \vec{Q}^L) \cup ((\vec{P} \times \vec{Q}^R) \cdot t)$
end if
Chapter 3

Main Results

In this chapter, we give 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 \( \sigma \) and \( C_n(\sigma) \) denotes the set of \( n \)-permutations that contain \( \sigma \). The algorithm for generating \( C_n(\sigma) \) is sketched as follows. First, we generate all numerical sequences satisfying the following conditions:

1. It is order isomorphic to \( \sigma \).
2. 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, ..., 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, ..., **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,
Step A \{\{23\}, \{24\}, \{342\}, \{234\}\}

Step B \{3124, 4123, 4132, 4231\}

Step C \{3124, 3142, 3412, 4312, 4123, 4132, 4213, 2413, 4231, 4213, 4123, 1423\}

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 $\sigma$.

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

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 $\sigma$. All possible numerical sequences order isomorphic to $\sigma$ appear in the $l$-prefix of the permutations which are generated in step B. The distribution by step C generates all permutations $\pi$ such that at least one of $\binom{n}{l}$ subsequences in $\pi$ exactly matches one of the numerical sequences order isomorphic to $\sigma$. They may be some duplications. But we do not have to consider the duplications when using $\pi$DDs because $\pi$DDs 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 $\pi$DDs as shown in Section 2.3.2. Let $A$ denote the $\pi$DD for permutations which are generated in step A, and let $B$ and $C$ denote the $\pi$DDs for the rearrangements which correspond to steps B and C, respectively. Note that the permutations represented in $B$ are not the permutations obtained after step B by rearranging those in $A$. The permutations in $B$ are the permutations as operations to apply those in $A$. The same applies to $C$. Then, $C_\pi(\sigma)$ can be obtained by computing $C \times B \times A$ (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 $B$ to $A$ and then apply $C$ to the result of the first application.

We show the method for the construction of $A$ last because it is similar to the construction of $C$ but more complicated.
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 $\sigma$, we define $\mathbb{B}$ to be the $\pi$DD consisting only of the permutation whose $l$-prefix is $\sigma$ and $(n-l)$-suffix is fixed. To construct this $\pi$DD, we first decompose $\sigma$ into a product of transpositions as given in Section 2.1. The $\pi$DD 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 $\pi$DD for the set of $n$-permutations $\pi$ such that there are $l$ indexes $1 \leq p_1 < p_2 < \ldots < p_l \leq n$ 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 $\pi$DD like the construction of $\mathbb{B}$. And then, we take the union of the $\pi$DDs. 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 $\binom{n}{l} = \binom{n-1}{l-1} + \binom{n-1}{l}$. Let $\text{Pos}_{i,j}$ be the set of all $n$-permutations $\pi$ containing at least one subsequence $\pi_{i_1}\pi_{i_2}\ldots\pi_{i_j}$ satisfying the following two conditions:

1. $1 \leq l_1 < l_2 < \ldots < l_j \leq i$.
2. $\pi_{i_1}\pi_{i_2}\ldots\pi_{i_j} = 12\ldots j$.

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

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

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

$p = 4, \sigma = 312, l = 3$

$$
\mathbb{C} \times \mathbb{B} \times \mathbb{A} = C_4(312)
$$
Here, we can partition $\text{Pos}_{i,j}$ into the two sets: the set including $\pi$ with $\pi_i \neq j$ and the other set. The former set equals $\text{Pos}_{i-1,j}$, and the latter one can be obtained by assigning $j$ to the $i$th position of permutations in $\text{Pos}_{i-1,j-1}$. Hence, if $\mathcal{P}_{i,j}$ denotes the $\pi$DD for $\text{Pos}_{i,j}$, this is achieved by $\mathcal{P}_{i-1,j-1} \cdot \tau_{i,j}$ because the $i$th element is $i$ from the third condition and $j$ is not assigned yet. Thus, $\mathcal{P}_{i,j} = \mathcal{P}_{i-1,j-1} \cup \mathcal{P}_{i-1,j-1} \cdot \tau_{i,j}$ holds. The dynamic programming for this recursion is encoded in Algorithm 3.

**Algorithm 3** Construct $\mathcal{C}$.

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

### 3.1.3 Construction of $\mathcal{A}$

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

We can obtain $\mathcal{A}$ by repeating assignments in a similar way to the construction of $\mathcal{C}$. However, this does not work if we assign elements in increasing order. In order to assign $v$ to $p$th position by $\tau_{p,v}$, $p$ must be fixed to $p$th position. But given that $p \leq v$, other transpositions could result in $p$ being in another position. In the construction of $\mathcal{P}_{p,v}$, there is no such problem because after $\tau_{p,v}$ assigns $v$ to $p$th position, there are no further assignments to $v$th 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 $\text{Inc}_{i,j}$ 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 $\pi$ in $\text{Inc}_{i,j}$, the $(l-j)$-suffix of the $l$-prefix of $\pi$ is already used and the $j$-prefix of the $l$-prefix of $\pi$ is fixed. We execute the constructions from $\text{Inc}_{n,l} = \{e_n\}$ to $\text{Inc}_{0,0}$. If $\mathcal{I}_{i,j}$ denotes the $\pi$DD for $\text{Inc}_{i,j}$, then
$$\mathbb{I}_{i,j} = \mathbb{I}_{i+1,j} \cup \mathbb{I}_{i+1,j+1} \cdot \mathbf{c}_{i+1,j+1}$$ holds because \( \mathbb{I}_{i,j} \) can be partitioned into the set including \( \pi \) with \( \pi_{j+1} = i+1 \), which is \( \mathbb{I}_{i+1,j+1} \cdot \mathbf{c}_{i+1,j+1} \) and the other set, which is \( \mathbb{I}_{i,j} \), 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 \( \text{suf}_{n,l} \times \mathbb{I}_{0,0} \), where \( \text{suf}_{n,l} \) is the \( \pi \text{DD} \) for the set including the \( n \)-permutations \( \pi \) in which \( \pi_i = i \) holds for \( 1 \leq i \leq l \) and the \((n-l)\)-suffix is in any order. We can obtain \( \text{suf}_{n,l} \) 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 \( \mathbb{A} \).

\[
\text{for } i = n - 1 \text{ to } 0 \text{ do} \\
\text{for } j = l \text{ to } 0 \text{ do} \\
\quad \text{if } j < l \text{ then} \\
\quad \quad \mathbb{I}_{i,j} = \mathbb{I}_{i+1,j} \cup \mathbb{I}_{i+1,j+1} \cdot \mathbf{c}_{i+1,j+1} \\
\quad \text{else} \\
\quad \quad \mathbb{I}_{i,j} = \mathbb{I}_{i+1,j} \\
\quad \text{end if} \\
\text{end for} \\
\text{end for} \\
\text{sufl} \leftarrow \pi \text{DD for } \{e_n\} \\
\text{for } i = l + 1 \text{ to } n \text{ do} \\
\quad \text{suf}_{i,l} = \text{suf}_{i-1,l} \\
\quad \text{for } j = l + 1 \text{ to } i - 1 \text{ do} \\
\quad \quad \text{suf}_{i,l} \leftarrow \text{suf}_{i,l} \cup \text{suf}_{i-1,l} \cdot \mathbf{c}_{i,j} \\
\quad \text{end for} \\
\text{end for} \\
\text{return } \text{suf}_{n,l} \times \mathbb{I}_{0,0}
\]

### 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′. C′ denotes the \( \pi \text{DD} \) 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 C′. In other words, if we assign \( j \) to \( i \)th position, we must assign \((j+1)\) to \((i+1)\)th position. For C′, we define \( \mathbb{P}_{i,j} = \mathbb{P}_{i-1,j} \cdot \mathbf{c}_{i,j} \) if the \( j \)th and the \((j+1)\)th elements must be adjacent, and otherwise \( \mathbb{P}_{i,j} = \mathbb{P}_{i-1,j} \cup \mathbb{P}_{i-1,j-1} \cdot \mathbf{c}_{i,j} \) as \( \mathbb{P}_{i,j} \).
As shown in Section 3.1.2, $P'_{i-1,j}$ consists only of permutations $\pi$ such that $\pi_i \neq j$, and $P'_{i-1,j-1} \cdot \tau_{i,j}$ consists only of $\pi$ such that $\pi_i = j$. Thus, if the $j$th and the $(j+1)$th elements must be adjacent, $P'_{i,j}$ includes only permutations $\pi$ such that $\pi_i = j$, and $P'_{i,j} \cdot \tau_{i+1,j+1}$ includes only permutations $\pi$ such that $\pi_i = j$ and $\pi_{i+1} = j+1$, that is, $j$ and $j+1$ are adjacent. Therefore, if the $j$th and the $(j+1)$th elements must be adjacent, $P'_{i,j}$ includes only permutations $\pi$ such that $\pi_i = j$ and $\pi_{i+1} = j+1$, that is, $j$ and $j+1$ are adjacent. Therefore, we obtain $P'_{n,l}$ by Algorithm 5, which is Algorithm 3 with one additional branch.

Algorithm 5

Construct $C'$.

\[ P'_{0,0} \leftarrow \pi \text{DD } \{e_n\} \]

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

$P'_{i,j} \leftarrow P'_{i-1,j-1} \cdot \tau_{i,j}$

else

$P'_{i,j} \leftarrow P'_{i-1,j-1} \cdot \tau_{i,j}$

end if

else

$P'_{i,j} \leftarrow P'_{i-1,j}$

end if

end for

end for

return $P'_{n,l}$

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 $j$th and the $(j+1)$th values must be consecutive, we define $I'_{i,j} = I'_{i+1,j+1} \cdot \tau_{i+1,j+1}$. 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 $\sigma$-avoiding $n$-permutations. This algorithm makes use of the following fact: the set of $\sigma$-avoiding permutations is the complement of the set of permutations that contain $\sigma$. Hereafter, $Av_n(\sigma)$ denotes the set of
Algorithm 6 Construct $\mathcal{A}'$.

\[
\begin{align*}
&\Pi_{n,j} \leftarrow \piDD \text{ for } \{e_n\} \\
&\text{for } i = n - 1 \text{ to } 0 \text{ do} \\
&\quad \text{for } j = l \text{ to } 0 \text{ do} \\
&\quad \quad \text{if } j < l \text{ then} \\
&\quad \quad \quad \text{if } j\text{th and }(j+1)\text{th elements must be consecutive then} \\
&\quad \quad \quad \quad \Pi_{i,j} \leftarrow \Pi_{i+1,j+1} \cdot \tau_{i+1,j+1} \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad \quad \Pi_{i,j} \leftarrow \Pi_{i+1,j} \cup \Pi_{i+1,j+1} \cdot \tau_{i+1,j+1} \\
&\quad \quad \quad \text{end if} \\
&\quad \quad \text{else} \\
&\quad \quad \quad \Pi_{i,j} \leftarrow \Pi_{i+1,j} \\
&\quad \quad \text{end if} \\
&\quad \text{end for} \\
&\text{end for} \\
&\text{end for} \\
&\text{for } i = l + 1 \text{ to } n \text{ do} \\
&\quad suf_{i,l} \leftarrow \piDD \text{ for } \{e_n\} \\
&\quad suf_{i,j} = suf_{i-1,j} \\
&\quad \text{for } j = l + 1 \text{ to } i - 1 \text{ do} \\
&\quad \quad suf_{i,j} \leftarrow suf_{i,j} \cup suf_{i-1,j} \cdot \tau_{i,j} \\
&\quad \text{end for} \\
&\text{end for} \\
&\text{return } suf_{n,l} \times \Pi_{0,0}
\end{align*}
\]

$\sigma$-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 $\sigma$ 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 $\piDD$ can be efficient because it depends on the size of the $\piDD$s. We now introduce the algorithm for generating $S_n$.

3.4.1 Generation Algorithm for all $n$-Permutations

Let $S_n$ denote the $\piDD$ for $S_n$. We can recursively construct $S_n$. Suppose we obtained $S_{n-1}$. We consider $(n-1)$-permutations as $n$-permutations with $\pi_n = n$. Thus, $S_{n-1} \cdot \tau_{1,n}$ consists of all $n$-permutations $\pi$ such that $\pi_k = k$. Therefore, $S_n$ can be obtained by computing $S_{n-1} \cdot \tau_{1,n} \cup S_{n-1} \cdot \tau_{2,n} \cup \ldots \cup S_{n-1} \cdot \tau_{n-1,n} \cup S_{n-1}$. Algorithm 7 realizes this procedure, using loops. Figure 3.3 shows $S_4$. While the cardinality of $S_n$ is $n!$, the size of $S_n$ is $O(n^2)$ as shown in Figure 3.3. This demonstrates a high compression ratio of $\piDD$s.
Algorithm 7 Construct $S_n$.

$S_0 \leftarrow \pi\text{DD for } \{e_n\}$

for $i = 1$ to $n$ do

$S_i \leftarrow S_{i-1}$

for $j = 1$ to $i - 1$ do

$S_i \leftarrow S_i \cup (S_{i-1} \setminus \tau_{i,j})$

end for

end for

return $S_n$

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 $\pi\text{DD}$ for $S_n$. Next, we construct the $\pi\text{DD}$ 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.
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 generate $P_{n,\sigma}$, and then computes $F_n^{(k)}(\sigma) = \{ \pi | P_{n,\sigma}(\pi) = k \}$ from $P_{n,\sigma}$.

3.5.1 Generation Algorithm for $P_{n,\sigma}$

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

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 $\pi$DD, such duplications are not counted. But by using the Cartesian product of $\pi$DD vectors instead of $\pi$DDs, we can generate the multiset of permutations whose multiplicity represents the pattern occurrence count, i.e. $P_{n,\sigma}$.

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

We can obtain $P_{n,\sigma}$ by using Cartesian product of $\pi$DD vectors as described in the previous subsection. However, in order to computationally check whether or not strong Wilf-equivalence between $\sigma_1$ and $\sigma_2$ holds, we must calculate the cardinalities of $F_n^{(k)}(\sigma_1)$ and $F_n^{(k)}(\sigma_2)$ for $0 \leq k \leq M$, where $M$ denotes the maximum number of occurrences of a given pattern. Note that $M \leq \binom{n}{l}$ 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 $\pi$DDs for $F_n^{(k)}(\sigma)$ for $0 \leq k \leq M$ from the $\pi$DD vector for $P_{n,\sigma}$ which was constructed by the algorithm in the previous subsection.

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

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

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

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

24
We introduce don’t care * to the binary representation of integers. Here, \( W_{(\ldots \ast 0k_i \ldots k_0)} \cup \) \( W_{(\ldots \ast 1k_i \ldots k_0)} = W_{(\ldots \ast 0k_i \ldots k_0)} \) and \( W_{(\ldots \ast 0k_i \ldots k_0)} \cap \) \( W_{(\ldots \ast 1k_i \ldots k_0)} = \emptyset \) hold. Hence,

\[
\begin{align*}
W_{(\ldots \ast 1k_i \ldots k_0)} &= W_{(\ldots \ast 0k_i \ldots k_0)} \cap P_{i+1}, \\
W_{(\ldots \ast 0k_i \ldots k_0)} &= W_{(\ldots \ast 0k_i \ldots k_0)} \setminus W_{(\ldots \ast 1k_i \ldots k_0)}.
\end{align*}
\]

Therefore, we can calculate \( W_k \) for \( 0 \leq k \leq M \) from \( W_{(\ldots \ast 0)} = S_n \) 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 \( \pi \)DD operation. Therefore, we can generate \( F_n^{(k)}(\sigma) \) for \( 0 \leq k \leq M \) by \( O(M) \) \( \pi \)DD operations based on the recursions. Algorithm 8 gives a pseudo code of this algorithm. This algorithm temporarily uses \( W_{(\ldots \ast k_i \ldots k_0)} \) to represent \( W_{(\ldots \ast 0k_i \ldots k_0)} \).

**Algorithm 8** Generate \( W_k \) for all \( 0 \leq k \leq M \) from \( \tilde{P} = (P_0, P_1, \ldots, P_m) \).

\[
W_0 \leftarrow \pi \text{DD for } S_n
\]

for \( i = 0 \) to \( m \) do

for \( \text{bin } = 2^i \) to \( 2^{i+1} - 1 \) do

\[
\begin{align*}
W_{\text{bin}} &\leftarrow W_{\text{bin}-2} \cap P_i, \\
W_{\text{bin}-2} &\leftarrow W_{\text{bin}-2} \setminus W_{\text{bin}}
\end{align*}
\]

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 \( \pi \)DDs for \( A, B, \) and \( \mathbb{C} \) by the algorithms described in Section 3.1. Second, we transform these \( \pi \)DDs into the \( \pi \)DD vectors, that is, \( \tilde{P} = (\mathbb{C}) \) and \( \tilde{Q} = (B \times A) \). Third, we calculate the Cartesian product \( \tilde{P} \times \tilde{Q} \), which is the \( \pi \)DD vector for \( P_{n,\sigma} \). Finally, we calculate the \( \pi \)DDs for \( F_n^{(k)}(\sigma) \) from the \( \pi \)DD vector for \( P_{n,\sigma} \) using Algorithm 8. This process is illustrated in Figure 3.5.

![Figure 3.5. The summary of our algorithm for \( F_n^{(k)}(\sigma) \).](image-url)
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 \( \sigma \) by checking the order isomorphism between all \( k \)-subsequences and \( \sigma \).

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \pi )DD Method</th>
<th>Naive Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>best</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>average</td>
<td>0.000</td>
<td>0.002</td>
</tr>
<tr>
<td>worst</td>
<td>0.000</td>
<td>0.008</td>
</tr>
<tr>
<td>9</td>
<td>best</td>
<td>0.000</td>
</tr>
<tr>
<td>average</td>
<td>0.004</td>
<td>0.005</td>
</tr>
<tr>
<td>worst</td>
<td>0.008</td>
<td>0.016</td>
</tr>
<tr>
<td>10</td>
<td>best</td>
<td>0.004</td>
</tr>
<tr>
<td>average</td>
<td>0.008</td>
<td>0.011</td>
</tr>
<tr>
<td>worst</td>
<td>0.012</td>
<td>0.020</td>
</tr>
<tr>
<td>11</td>
<td>best</td>
<td>0.000</td>
</tr>
<tr>
<td>average</td>
<td>0.008</td>
<td>0.029</td>
</tr>
<tr>
<td>worst</td>
<td>0.016</td>
<td>0.052</td>
</tr>
<tr>
<td>12</td>
<td>best</td>
<td>0.004</td>
</tr>
<tr>
<td>average</td>
<td>0.014</td>
<td>0.087</td>
</tr>
<tr>
<td>worst</td>
<td>0.024</td>
<td>0.156</td>
</tr>
<tr>
<td>13</td>
<td>best</td>
<td>0.016</td>
</tr>
<tr>
<td>average</td>
<td>0.022</td>
<td>0.284</td>
</tr>
<tr>
<td>worst</td>
<td>0.022</td>
<td>0.544</td>
</tr>
<tr>
<td>14</td>
<td>best</td>
<td>0.008</td>
</tr>
<tr>
<td>average</td>
<td>0.032</td>
<td>1.029</td>
</tr>
<tr>
<td>worst</td>
<td>0.056</td>
<td>1.968</td>
</tr>
<tr>
<td>15</td>
<td>best</td>
<td>0.016</td>
</tr>
<tr>
<td>average</td>
<td>0.052</td>
<td>3.513</td>
</tr>
<tr>
<td>worst</td>
<td>0.088</td>
<td>6.860</td>
</tr>
</tbody>
</table>
Table 4.2. Memory consumption (kB) for generating classical pattern-avoiding permutations.

<table>
<thead>
<tr>
<th>n</th>
<th>(Av_n(σ))</th>
<th>DD Method</th>
<th>Naive Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>best</td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>8</td>
<td>1430</td>
<td>1430</td>
<td>1430</td>
</tr>
<tr>
<td>9</td>
<td>462</td>
<td>462</td>
<td>462</td>
</tr>
<tr>
<td>10</td>
<td>1679</td>
<td>1679</td>
<td>1679</td>
</tr>
<tr>
<td>11</td>
<td>5878</td>
<td>5878</td>
<td>5878</td>
</tr>
<tr>
<td>12</td>
<td>20812</td>
<td>20812</td>
<td>20812</td>
</tr>
<tr>
<td>13</td>
<td>742900</td>
<td>742900</td>
<td>742900</td>
</tr>
<tr>
<td>14</td>
<td>2674440</td>
<td>2674440</td>
<td>2674440</td>
</tr>
<tr>
<td>15</td>
<td>9694845</td>
<td>9694845</td>
<td>9694845</td>
</tr>
</tbody>
</table>

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 \( \pi \)DD 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 \( \pi \)DD 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 \( \pi \)DD method is greater than naive method, but it is very smaller than the estimate especially when \( l \) is near \( n \). This shows that \( \pi \)DD 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 \( \pi \)DDs. 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, \( 3142 \) and \( 2413 \), and appear in many mathematical problems [1, 8, 10]. Our algorithm generates \( \text{Av}_n(3142, 2413) = S_n \setminus (C_n(3142) \cup C_n(2413)) \). 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.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$B(n)$</th>
<th>$\pi$DD Method</th>
<th>Naive Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time (s)</td>
<td>Memory (kB)</td>
</tr>
<tr>
<td>8</td>
<td>10754</td>
<td>0.016</td>
<td>2760</td>
</tr>
<tr>
<td>9</td>
<td>58202</td>
<td>0.028</td>
<td>4164</td>
</tr>
<tr>
<td>10</td>
<td>326240</td>
<td>0.060</td>
<td>12984</td>
</tr>
<tr>
<td>11</td>
<td>1882960</td>
<td>0.212</td>
<td>26828</td>
</tr>
<tr>
<td>12</td>
<td>11140560</td>
<td>0.908</td>
<td>98924</td>
</tr>
<tr>
<td>13</td>
<td>67329992</td>
<td>3.678</td>
<td>383272</td>
</tr>
<tr>
<td>14</td>
<td>414499438</td>
<td>13.419</td>
<td>824732</td>
</tr>
<tr>
<td>15</td>
<td>2593341586</td>
<td>50.499</td>
<td>3151164</td>
</tr>
<tr>
<td>16</td>
<td>16458756586</td>
<td>193.704</td>
<td>12403488</td>
</tr>
<tr>
<td>17</td>
<td>105791986682</td>
<td>745.779</td>
<td>40788188</td>
</tr>
</tbody>
</table>

Table 4.4. The $\pi$DD for Baxter permutations ($n = 15$).

<table>
<thead>
<tr>
<th></th>
<th>#permutations</th>
<th>#nodes in $\pi$DD</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_n$</td>
<td>1307674368000</td>
<td>105</td>
</tr>
<tr>
<td>$C_n(3142) \cup C_n(2413)$</td>
<td>1305081026414</td>
<td>4094585</td>
</tr>
<tr>
<td>$Av_n(3142, 2413)$</td>
<td>2593341586</td>
<td>2158472</td>
</tr>
</tbody>
</table>

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 $\pi$DDs is not efficient, but, $\pi$DD’s difference operation is not a bottleneck in this problem. Most of the computation time is due to the Cartesian product operations between three $\pi$DDs, which require 46.020 seconds.

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

We show the results for bivincular patterns. Specifically, we generated $Av_n(T_3^{23})$, 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 $Av_n(T_3^{23})$. 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 $Av_n(T_3^{23})$ are all decreasing. For $n = 13$, the $\pi$DD 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.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( A_{n} (123) ) ( (231) )</th>
<th>( \pi \text{DD Method} )</th>
<th>( \text{Naive Method} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Memory (kB)</td>
<td>Time (s)</td>
</tr>
<tr>
<td>8</td>
<td>5335</td>
<td>0.004</td>
<td>3084</td>
</tr>
<tr>
<td>9</td>
<td>31240</td>
<td>0.004</td>
<td>4196</td>
</tr>
<tr>
<td>10</td>
<td>201608</td>
<td>0.020</td>
<td>12948</td>
</tr>
<tr>
<td>11</td>
<td>1422074</td>
<td>0.052</td>
<td>26436</td>
</tr>
<tr>
<td>12</td>
<td>10886503</td>
<td>0.184</td>
<td>98656</td>
</tr>
<tr>
<td>13</td>
<td>89903100</td>
<td>0.916</td>
<td>382712</td>
</tr>
<tr>
<td>14</td>
<td>796713190</td>
<td>3.808</td>
<td>382712</td>
</tr>
<tr>
<td>15</td>
<td>7541889195</td>
<td>15.357</td>
<td>1510640</td>
</tr>
<tr>
<td>16</td>
<td>75955177642</td>
<td>62.408</td>
<td>3327380</td>
</tr>
<tr>
<td>17</td>
<td>810925547354</td>
<td>254.896</td>
<td>12887884</td>
</tr>
<tr>
<td>18</td>
<td>9148832109645</td>
<td>1016.132</td>
<td>50354060</td>
</tr>
</tbody>
</table>

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 \( \pi \text{DD} \) method only include the constructions of \( \pi \text{DDs} \), meaning that we do not decompress \( \pi \text{DDs} \) 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 \( \pi \text{DD} \) depends on the size of \( \pi \text{DD} \), while that of the naive method depends on the number of subsequences \( \binom{n}{l} \). In practical cases, the size of a \( \pi \text{DD} \) 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 \( \sigma \) 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 \( \pi \text{DDs} \) 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 \( \pi \text{DDs} \) is proportional to the size of \( \pi \text{DDs} \). 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 \( \pi \text{DDs} \) for \( F_{n}^{(k)} (\sigma) \) is smaller than the number of all \( n \)-permutations \( n! \). Moreover, since some \( \pi \text{DDs} \) share their equivalent subgraphs, the actual number of the total nodes of the \( \pi \text{DDs} \) on memory may be smaller than the results in Table 4.7. This shows that \( \pi \text{DDs} \) achieve compact representations of \( F_{n}^{(k)} (\sigma) \).
Table 4.6. Computation time (sec) for generating $F_n^{(k)}(\sigma)$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\pi$DD Method</th>
<th>Naive Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$l$</td>
<td>$l$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>best</td>
<td>0.032</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>0.036</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>0.040</td>
</tr>
<tr>
<td>9</td>
<td>best</td>
<td>0.076</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>0.078</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>0.080</td>
</tr>
<tr>
<td>10</td>
<td>best</td>
<td>0.276</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>0.310</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>0.344</td>
</tr>
<tr>
<td>11</td>
<td>best</td>
<td>1.436</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>1.642</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>1.848</td>
</tr>
<tr>
<td>12</td>
<td>best</td>
<td>6.316</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>7.064</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>7.812</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>memory consumption (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$l$</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>average</td>
</tr>
<tr>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>9</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>average</td>
</tr>
<tr>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>10</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>average</td>
</tr>
<tr>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>11</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>average</td>
</tr>
<tr>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>12</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>average</td>
</tr>
</tbody>
</table>

Table 4.8 provides candidates of strongly Wilf-equivalent classes. Two patterns $\sigma_1$ and $\sigma_2$ 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 $\pi$ and its reverse $\pi^r$ are always strongly Wilf-equivalent. Note that a pattern $\pi$ and its inverse $\pi^{-1}$ 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.

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

Conclusion

In this thesis, we introduced algorithms for generating several pattern-avoiding permutations using $\pi$DDs. 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 $\pi$DDs 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 $P_{n,\sigma}$ for given $n$ and $\sigma$ using $\pi$DD vectors, and an algorithm computing the $\pi$DDs representing $F_n^{(k)}(\sigma)$ for each $k$ from the $\pi$DD vector for $P_{n,\sigma}$. 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 $\pi$DDs 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


