TCS-TR-A-13-67

TCS Technical Report

Implicit Generation of Pattern-Avoiding Permutations Based on πDD

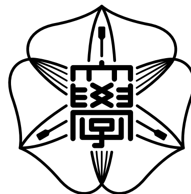
by

Yuma Inoue, Takahisa Toda and Shin-ichi Minato

Division of Computer Science

Report Series A

September 21, 2013



Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682

Fax: +81-011-706-7682

Implicit Generation of Pattern-Avoiding Permutations Based on πDDs

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

Takahisa Toda
JST ERATO Minato Project
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

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

September 21, 2013

Abstract

Pattern-avoiding permutations are permutations where none of the subsequences match the relative order of a given pattern. Pattern-avoiding permutations 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 design, can be encoded into pattern-avoiding permutations. The generation of pattern-avoiding permutations is an important topic in efficient VLSI design and mathematical analysis of patten-avoiding permutations. In this paper, we present an algorithm for generating pattern-avoiding permutations, and extend this algorithm beyond classical patterns to generalized patterns with more restrictions. Our approach is based on the data structure πDDs, which can represent a permutation set compactly and has useful set operations. We demonstrate the efficiency of our algorithm by computational experiments.

1 Introduction

A permutation π *avoids* a *pattern* σ if no subsequence in π 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 the rule $a_i < a_j$ if and only if $b_i < b_j$ for all i, j. Permutations that avoid pattern σ are called σ-*avoiding* permutations.

*He also works for JST ERATO Minato Project.

Research of pattern-avoiding permutations dates back to *stack sort*, which was proposed by Knuth in [10]. 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 [9], and the double-ended queue sorting problem [15], have been proposed, and pattern-avoiding permutations were developed in that context.

After pattern-avoiding permutations were proposed, many researchers engaged in studies to compute the number of the permutations that avoid given patterns. For example, 1342-avoiding permutations have been enumerated by a mathematical approach [3], and 1324-avoiding permutations can be counted by computer programs [12]. Moreover, the relation of classes on pattern-avoiding permutations has also been examined. Two classes A_1 and A_2 are *Wilf-equivalent* if $|A_1| = |A_2|$, where $|S|$ denotes the cardinality of set S. In [16], the nontrivial Wilf-equivalence between 4132-avoiding and 3142-avoiding was discovered. The generation of pattern-avoiding permutations can contribute to not only the discovery of unknown Wilf-equivalent classes, but also the identification of bijective functions between such classes.

Relations between pattern-avoiding permutations and mathematical problems have been studied actively [6, 7]. In particular, Yao et al. revealed a bijection between *mosaic floorplans* and *Baxter permutations*, which are generalized pattern-avoiding permutations [18], and Ackerman et al. proposed a simple encoding and decoding between them in [1]. A floorplan is a topological partition of a rectangle into multiple rectangles, and a mosaic floorplan is a subclass of floorplans. Floorplans have practical applications in areas such as VLSI design. Storing all pattern-avoiding permutations into a database is equivalent to preparing a database of floorplans. Database queries such as searching by criteria and random sampling are useful for VLSI design. Therefore, generating pattern avoiding permutations can contribute to solving practical problems.

Wilf provided a generating algorithm for identity patterns and posed the question about the complexity of generation of pattern-avoiding permutations for general patterns [17]. In [4], Bose et al. proved that the permutation pattern matching problem is NP-complete and the enumerating problem is #P-complete. They also proposed a decision algorithm for the special case when the pattern is a separable permutation. The complexity of this algorithm is $O(kn^6)$, which Ibarra[8] improved to $O(kn^4)$, where n is the permutation length and k is the pattern length. However, as far as we know, no generating algorithm for general patterns except for brute force method has been proposed.

In this paper, we provide an efficient algorithm for generating pattern-avoiding permutations. Furthermore, we extend our algorithm to handle some generalized patterns, such as vincular patterns and bivincular patterns.

Our algorithm is based on a permutation decision diagram (πDD), which is a data structure for compact representation of sets of permutations [14]. A πDD not only achieves high compression of sets of permutations, but also supports 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. If the πDD is small, computation is fast especially when the number of permutations is large.

The rest of this paper is organized as follows. Section 2 presents the notations and definitions of permutations and patterns. Section 3 describes the basics of πDDs. Section 4 presents our algorithm for generating pattern-avoiding permutations and its extension to some generalized patterns. Section 5 presents experimental results, and Section 6 concludes this paper.

2 Permutations and Patterns

2.1 Permutations

A permutation of length n (n-permutation for brevity) is a bijection from $\{1, 2, ..., n\}$ to itself. Let π be an n-permutation. We write a permutation in the one-line form as $\pi = \pi_1\pi_2 \dots \pi_n$, and denote $i\pi = \pi_i$. For example, $\pi = 4312$ is a 4-permutation and $3\pi = 1$.

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

In this paper, we use the multiplication $x \cdot y$ in order to permutes 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 $i\tau_{(i,j)} = j$, $j\tau_{(i,j)} = i$, and $k\tau_{(i,j)} = k$ 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 [11]. 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 (Fig. 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 pattern σ if there is at least one subsequence in π which is order isomorphic to σ, where the subsequence need not consist of consecutive

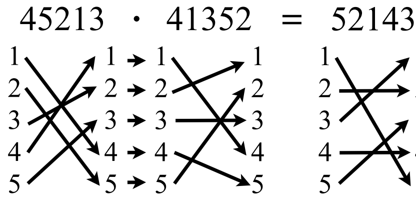


Figure 1: An example of the multiplication of permutations.

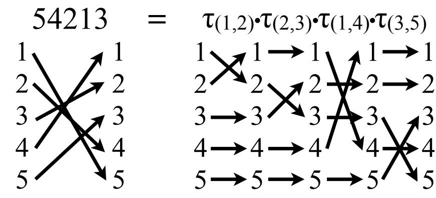


Figure 2: Decomposition of 54213 into transpositions.

numbers in π. In other words, let k be the length of σ. There are indexes $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ such that $\pi_{i_x} < \pi_{i_y}$ if and only if $\sigma_x < \sigma_y$, for all pairs of x and y. For example, the permutation 4213 contains 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 *generalized pattern*, is a well-known generalization [2]. 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* [5]. A bivincular pattern is restricted by adjacency of positions and additionally by consecutiveness of values. We use the 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 element 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 problem considered in this paper can be stated as follows: for given a positive integer n and a pattern σ, generate all σ-avoiding permutations of length n.

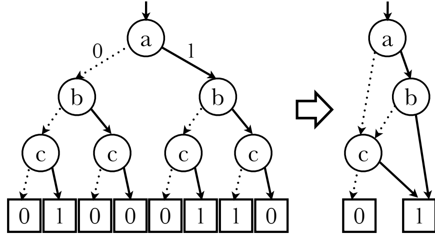


Figure 3: A binary decision tree and a ZDD for a set of combinations.

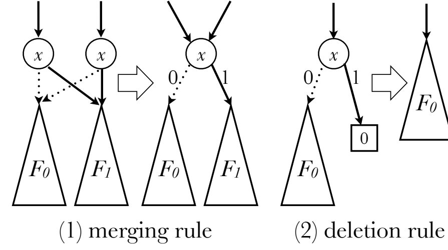


Figure 4: Two reduction rules on ZDDs.

3 πDDs

A πDD is a data structure which canonically represents and efficiently manipulates a set of permutations. The efficiency of our algorithm is based on the compact representation and rich set operations of πDDs. πDDs are based on *zero-suppressed binary decision diagrams* (*ZDDs*) [13], which are decision diagrams for sets of combinations (families of sets).

3.1 ZDDs

A ZDD is derived by reducing a binary decision tree. Figure 3 shows the ZDD for the family of sets $\{\{a, b\}, \{a, c\}, \{c\}\}$. A ZDD has five components: internal nodes with an item labels, 0-edges, 1-edges, the 0-terminal node, and the 1-terminal node. Each 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.

A ZDD is a compact and canonical form 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 child nodes 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 Fig. 4. In the worst-case scenario, the size of a ZDD (the number of nodes in a ZDD) can grow exponentially with respect to the number of items. In many practical cases, though, a ZDD provides 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 ZDDs, not on the cardinality of the sets represented by the ZDDs.

$P \cup Q$	Union $\{\pi \mid \pi \in P \text{ or } \pi \in Q\}$.
$P \setminus Q$	Difference $\{\pi \mid \pi \in P \text{ and } \pi \notin Q\}$.
$P.Swap(x, y)$	Swap $\{\pi \cdot \tau_{(x,y)} \mid \pi \in P\}$.
$P \times Q$	Cartesian product $\{\alpha \cdot \beta \mid \alpha \in P \text{ and } \beta \in Q\}$.

Table 1: πDD operations.

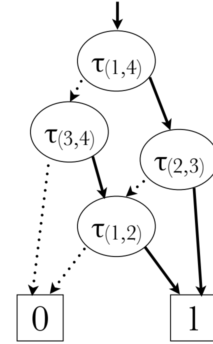


Figure 5: The πDD for $\{2143, 2431, 4321\} = \{\tau_{(1,2)} \cdot \tau_{(3,4)}, \tau_{(1,2)} \cdot \tau_{(1,4)}, \tau_{(2,3)} \cdot \tau_{1,4)}\}$.

3.2 πDDs

We introduce a πDD, which represents a set of permutations by deriving it from a ZDD. As shown in Sect. 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 a πDD. Since the transpositions $\tau_{(i,j)}$ into which a permutation is decomposed are ordered in increasing order of j, the order of transpositions in a πDD can be fixed. πDDs obtain a compact and canonical form by applying the above two reduction rules in the same way to ZDDs, and set operations are also available. Figure 5 shows an example of a πDD.

Table 1 shows the πDD operations used in this paper. While the union and set difference operations are available like ZDDs, the swap and Cartesian product operations are unique to πDDs. In particular, Cartesian product is very useful because multiplication over permutations results in rearrangement. That is, by applying Cartesian product operator, we can execute rearrangements of multiple numerical sequences at once.

4 Main Results

In this section, we propose 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 σ-avoiding n-permutations, S_n denotes the set of all n-permutations, and $C_n(\sigma)$ denotes the set of n-permutations that contain σ. As stated above, $Av_n(\sigma) = S_n \setminus C_n(\sigma)$ holds. The generation of the set of permutations which contain σ is easily possible by simple rearrangements as shown in

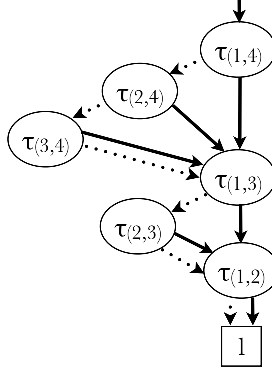


Figure 6: The πDD for S_4.

Sect. 4.2. In general, the time to compute set difference depends on the cardinalities of the sets. On the other hand, the set difference operation of πDD can be efficient because it depends on the size of the πDDs.

First, we introduce the algorithm for generating S_n. Next, we show the algorithm for generating $C_n(\sigma)$ for classical patterns. The algorithm can be easily extended to vincular and bivincular patterns, which is demonstrated in Sect. 4.3 and 4.4.

4.1 Generating 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 $n\pi = n$. Thus, $\mathbb{S}_{n-1}.Swap(k, n)$ consists of all n-permutations π such that $n\pi = k$. Therefore, $\mathbb{S}_n$ can be obtained by computing $\mathbb{S}_{n-1}.Swap(1, n) \cup \mathbb{S}_{n-1}.Swap(2, n) \cup \ldots \cup \mathbb{S}_{n-1}.Swap(n-1, n) \cup \mathbb{S}_{n-1}$. Algorithm 1 realizes this procedure by loops. Figure 6 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 Fig. 6. This is demonstrating a high compression ratio of πDDs.

Algorithm 1 Construct $\mathbb{S}_n$.

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

4.2 Generating permutations containing a classical pattern

Hereafter, unless otherwise noted, k denotes the length of a given pattern σ. In order to generate $C_n(\sigma)$, we assign all $\binom{n}{k}$ sequences that are order isomorphic to σ into $\binom{n}{k}$ possible positions in an n-permutation. This is achieved in three steps as follows:

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

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

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

Figure 7 shows the process of generating $C_4(312)$. Step 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 Sect. 3.2. Let $\mathbb{A}$ denote the πDD for permutations which generated in step A, and let $\mathbb{B}$ and $\mathbb{C}$ denote the πDDs for the rearrangements which correspond to step 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 $\mathbb{A}$.z The same applies to $\mathbb{C}$. Then, $C_n(\sigma)$ can be obtained by computing $\mathbb{C} \times \mathbb{B} \times \mathbb{A}$ (Fig. 8). 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 $\mathbb{A}$ at the end because it is similar to the construction of $\mathbb{C}$ but more complicated.

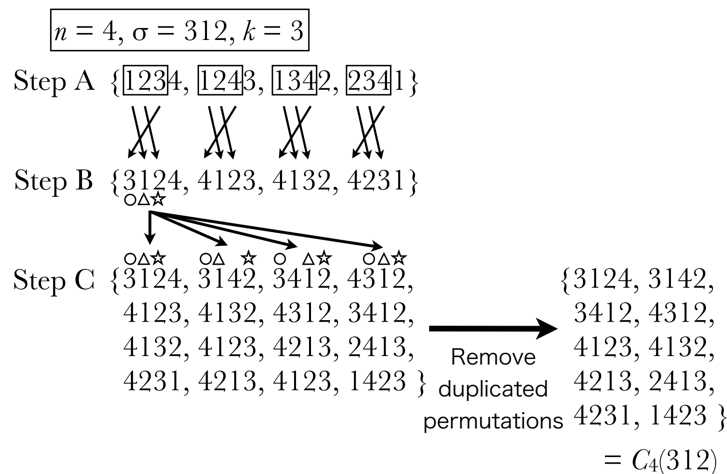


Figure 7: The process of generating $C_4(312)$.

$n = 4, \sigma = 312, k = 3$

$$\mathbb{C} \times \mathbb{B} \times \mathbb{A} = C_4(312)$$

$$\{1234, 1243, 1423, 4123\} \times \{3124\} \times \{1234, 1243, 1342, 2341\} = \{3124, 3142, 3412, 4312, 4123, 4132, 4213, 2413, 4231, 1423\}$$

Figure 8: Cartesian product in generating $C_4(312)$.

4.2.1 Construction of $\mathbb{B}$

$\mathbb{B}$ is the πDD consisting only of the permutation given as a pattern. To construct this πDD, we first decompose a given pattern into a product of transpositions as given in Sect. 2.1. The πDD for this decomposition can be easily constructed in a bottom-up fashion.

4.2.2 Construction of $\mathbb{C}$

$\mathbb{C}$ is the πDD for the set of n-permutations π such that there are k indexes $1 \leq p_1 < p_2 < \ldots < p_k \leq n$ with $p_i\pi = i$. This means that if π is in $\mathbb{C}$, the numerical sequence $12\ldots k$ must be in π as its subsequence. A simple method to construct $\mathbb{C}$ is as follows. We generate all n-permutations that satisfy the above condition, then convert each permutation to the πDD consisting only of the permutation. We then calculate the union of all these πDDs. This algorithm is simple and easy to implement. However, this is not efficient because this algorithm has to repeat union operations $\binom{n}{k}$ times.

Our basic idea to reduce the number of πDD operations is based on Pascal's triangle, in which the recursion $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ holds. Let $Pos_{i,j}$ be the set of all n-permutations π containing at least one subsequence $\pi_{k_1}\pi_{k_2}\ldots\pi_{k_l}$ satisfying the following two conditions:

1. k_l is less than or equal to i,
2. the subsequence matches the sequence $12\ldots j$.

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

1. For any two distinct members $\pi, \pi' \in Pos_{i,j}$ there is at least one index x such that $p_x \neq p'_x$.

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

Although the two conditions are not necessary to step C, these conditions simplify our algorithm. 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 into the ith position of permutations in $Pos_{i-1,j-1}$. Hence, let $\mathbb{P}_{i,j}$ denote the πDD for $Pos_{i,j}$, this is achieved by $\mathbb{P}_{i-1,j-1}.Swap(i,j)$ because the ith element is i from the second condition and j is not assigned yet. Thus, $\mathbb{P}_{i,j} = \mathbb{P}_{i-1,j} \cup \mathbb{P}_{i-1,j-1}.Swap(i,j)$ holds. The dynamic programming for this recursion is encoded in Algorithm 2.

Algorithm 2 Construct $\mathbb{C}$.

```
P_{0,0} ← πDD for {e_n}
for i = 1 to n do
  for j = 0 to k do
    if j > 0 then
      P_{i,j} ← P_{i−1,j} ∪ P_{i−1,j−1}.Swap(i, j)
    else
      P_{i,j} ← P_{i−1,j}
    end if
  end for
end for
return P_{n,k}
```

4.2.3 Construction of $\mathbb{A}$

Let $Inc_{i,j}$ denote the set of all i-permutations in which the j-prefix is ordered in increasing order. More precisely, $\pi \in Inc_{i,j}$ satisfies $1 \leq \pi_1 < \pi_2 < \ldots < \pi_j \leq i$.

$Inc_{i,j}$ can be obtained in a similar way to $Pos_{i,j}$. Let $\mathbb{I}_{i,j}$ denote the πDD for $Inc_{i,j}$. $Inc_{i,j}$ is dividable into two sets: the set including π with $\pi_j = i$ and the other set. Unfortunately, however, $\mathbb{I}_{i,j} \neq \mathbb{I}_{i-1,j} \cup \mathbb{I}_{i-1,j-1}.Swap(i,j)$ does not holds while $\mathbb{P}_{i,j} = \mathbb{P}_{i-1,j} \cup \mathbb{P}_{i-1,j-1}.Swap(i,j)$ holds. This difference is caused by the fact that $\pi \in Inc_{i-1,j-1}$ can be $\pi_j \neq j$. For example, we suppose that we want to obtain π such that $\pi_1 = 2$ and $\pi_2 = 3$, $\pi = \tau_{1,2} \cdot \tau_{2,3} = 312$ does not satisfy $\pi_2 = 3$ because $2\tau_{1,2} \neq 2$. This is avoidable by performing calculations in the decreasing order, for example, $\pi = \tau_{2,3} \cdot \tau_{1,2} = 231$. Thus, we start from $\mathbb{I}_{n,k}$ and end to $\mathbb{I}_{0,0}$ in the construction of $\mathbb{A}$.

There is one more difference from the construction of $\mathbb{P}_{i,j}$ as follows: the $(i-j)$-suffix of each permutation in $Inc_{i,j}$ is in any order. In other words, $|Inc_{i,j}| = \binom{i}{j} \cdot (i-j)!$. Hence, after we construct $Inc_{n,k}$, then rearrange $(n-k)$-suffix of each permutation in $Inc_{n,k}$ into any orders. The πDD for this rearrangement can be obtained by the construction like Algorithm 1. Algorithm 3 describes the entire process.

Algorithm 3 Construct $\mathbb{A}$.

```
I_{n,k} ← πDD for {e_n}
for i = n − 1 to 0 do
  for j = k to 0 do
    if j < k then
      I_{i,j} ← I_{i+1,j} ∪ I_{i+1,j+1}.Swap(i, j)
    else
      I_{i,j} ← I_{i+1,j}
    end if
  end for
end for

suf_k ← πDD for {e_n}
for i = k + 1 to n do
  suf_i = suf_{i−1}
  for j = k + 1 to i − 1 do
    suf_i ← suf_i ∪ suf_{i−1}.Swap(i, j)
  end for
end for
return suf_n × I_{0,0}
```

4.3 Generating permutations containing a vincular pattern

The additional restriction of vincular patterns is adjacency of positions. Therefore, we can generate vincular pattern-avoiding permutations by a slight modification of step C from Sect. 4.2.2. We call the modified step C'. $\mathbb{C}'$ denotes the πDD which correspond to step C'.

If the jth and the $(j-1)$th elements must be adjacent, we define $\mathbb{P}_{i,j} = \mathbb{P}_{i-1,j-1}.Swap(i,j)$, which is not united $\mathbb{P}_{i-1,j}$. Hence,

$$\begin{aligned}\mathbb{P}_{i+1,j+1} &= \mathbb{P}_{i,j+1} \cup \mathbb{P}_{i,j}.Swap(i,j)\\ &= \mathbb{P}_{i,j+1} \cup \mathbb{P}_{i,j-1}.Swap(i-1,j-1).Swap(i,j)\end{aligned}$$

That is, $\pi \in \mathbb{P}_{i-1,j-1}.Swap(i-1,j-1).Swap(i,j)$ must be $\pi_{j-1} = i-1$ and $\pi_j = i$. $\mathbb{P}_{i,j+1}$ is dividable into $\mathbb{P}_{i-1,j+1}$ and $\mathbb{P}_{i-1,j-1}.Swap(i-2,j-1).Swap(i-1,j)$, in which $\pi_{j-1} = i-2$ and $\pi_j = i-1$, as in $\mathbb{P}_{i+1,j+1}$, and so forth. This recursive structure shows that $\pi \in \mathbb{P}_{i+1,j+1}$ must be $\pi_j = \pi_{j-1}+1$: the jth and the $(j-1)$th elements are adjacent. Therefore, we obtain Algorithm 4 for $\mathbb{C}'$ by adding a branch to Algorithm 2.

4.4 Generating permutations containing a bivincular pattern

Bivincular patterns have the three restrictions: a relative order, adjacencies of positions and consecutiveness of values. Hence, we use step C' in Sect. 4.3 and change step A to A' in the same way as C was changed to C'.

Algorithm 4 Construct $\mathbb{C}'$.

```
ℙ_{0,0} ← πDD {e_n}
for i = 1 to n do
  for j = 0 to k do
    if j > 0 then
      if j-th and (j − 1)-th elements must be adjacent then
        ℙ_{i,j} ← ℙ_{i−1,j−1}.Swap(i, j)
      else
        ℙ_{i,j} ← ℙ_{i−1,j} ∪ ℙ_{i−1,j−1}.Swap(i, j)
      end if
    else
      ℙ_{i,j} ← ℙ_{i−1,j}
    end if
  end for
end for
return ℙ_{n,k}
```

If the ith and the $(i+1)$th values must be consecutive, we define $\mathbb{I}_{i,j} = \mathbb{I}_{i+1,j+1}.Swap(i,j)$, as in C'. Algorithm 5 describes the process to obtain $\mathbb{A}'$.

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

```
𝕀_{n,k} ← πDD for {e_n}
for i = n − 1 to 0 do
  for j = k to 0 do
    if j < k then
      if j-th and (j + 1)-th element must be consecutive then
        𝕀_{i,j} ← 𝕀_{i+1,j+1}.Swap(i, j)
      else
        𝕀_{i,j} ← 𝕀_{i+1,j} ∪ 𝕀_{i+1,j+1}.Swap(i, j)
      end if
    else
      𝕀_{i,j} ← 𝕀_{i+1,j}
    end if
  end for
end for

suf_k ← πDD for {e_n}
for i = k + 1 to n do
  suf_i = suf_{i−1}
  for j = k + 1 to i − 1 do
    suf_i ← suf_i ∪ suf_{i−1}.Swap(i, j)
  end for
end for
return suf_n × 𝕀_{0,0}
```

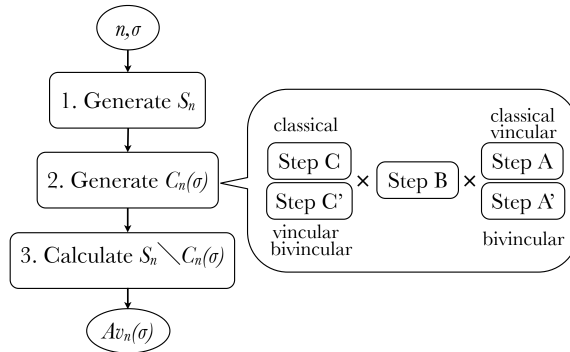


Figure 9: The summary of our algorithms.

4.5 Summary of our algorithms

Our algorithms can be summarized as the 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 Fig. 9.

5 Experimental Results

We implemented our algorithm in C++ and carried out computational experiments. We compared the performance of our algorithm to that of the naive method, which generates all n-permutations and, for each n-permutation, decides whether it contains σ by checking the order isomorphism between all k-subsequences and σ.

Tables 2 and 3 show the results for the generating permutations avoiding a classical pattern. The tables show the best, the worst, and the average computation time and memory consumption for generating all patterns with length k = 2,3,4, and 5. Note that the naive method both stores and outputs all pattern-avoiding permutations as a list of arrays. In almost all cases, our algorithm is more efficient than the naive method. For example, in $n = 11$, our method requires only 0.3% of the time and 1% of the memory 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 and memory consumption of both methods increase exponentially with respect to n, but our algorithm has a smaller growth rate than the naive method.

Table 3: Memory consumption (kB) for generating classical pattern-avoiding permutations.

n		πDD Method k				Naive Method k			
		2	3	4	5	2	3	4	5
8	best	1600	1596	1596	1596	1072	1332	2160	5756
	average	1600	1598	1916	1599	1072	1336	2162	5759
	worst	1600	1600	1972	1600	1072	1340	2164	5764
9	best	1600	1964	2748	2744	1072	1600	10316	19460
	average	1600	2233	2933	2768	1072	1600	10319	19463
	worst	1600	2768	4208	2792	1072	1600	10324	19468
10	best	1600	2760	4212	7156	1072	3476	74684	295940
	average	1784	3509	6856	7376	1072	3476	74686	295943
	worst	1968	4260	7444	7700	1072	3476	74688	295948
11	best	1600	4208	7676	13720	1072	6796	361444	2884764
	average	2186	5876	17641	25233	1072	6796	361447	2884767
	worst	2772	6792	25084	27000	1072	6796	361448	2884768
12	best	1976	7112	25316	49108	—	—	—	—
	average	3112	16106	48405	93525	—	—	—	—
	worst	4248	25084	94788	102940	—	—	—	—-
13	best	2764	12708	51436	188416	—	—	—	—
	average	4954	32130	168848	337123	—	—	—	—
	worst	7144	51084	201264	408460	—	—	—	—
14	best	2760	24440	187372	396432	—	—	—	—
	average	7754	111634	542621	1282547	—	—	—	—
	worst	12748	190460	779640	1587600	—	—	—	—
15	best	4228	47620	389140	1543108	—	—	—	—
	average	9112	234836	1560720	4986352	—	—	—	—
	worst	13996	406440	3092572	6471388	—	—	—	—

Table 2: Computation time (s) for generating classical pattern-avoiding permutations.

n		πDD Method k				Naive Method k			
		2	3	4	5	2	3	4	5
8	best	0.000	0.000	0.000	0.000	0.012	0.012	0.032	0.044
	average	0.000	0.002	0.004	0.001	0.014	0.017	0.038	0.050
	worst	0.000	0.008	0.012	0.008	0.016	0.024	0.044	0.064
9	best	0.000	0.000	0.004	0.004	0.056	0.120	0.420	0.732
	average	0.004	0.005	0.009	0.009	0.058	0.121	0.448	0.776
	worst	0.008	0.016	0.016	0.024	0.060	0.124	0.480	0.836
10	best	0.004	0.004	0.016	0.024	0.492	1.148	5.684	15.773
	average	0.008	0.011	0.028	0.036	0.494	1.170	6.135	16.970
	worst	0.012	0.020	0.044	0.060	0.496	1.184	6.596	18.525
11	best	0.000	0.012	0.044	0.092	5.420	13.129	87.946	409.694
	average	0.008	0.029	0.101	0.174	5.446	13.292	94.406	429.715
	worst	0.016	0.052	0.152	0.276	5.512	13.457	100.654	452.252
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	—	—	—	—

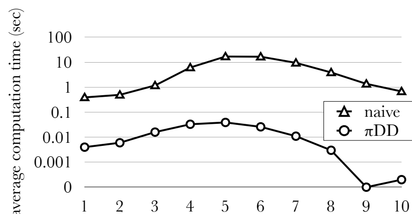


Figure 10: Computation time when $n = 10$.

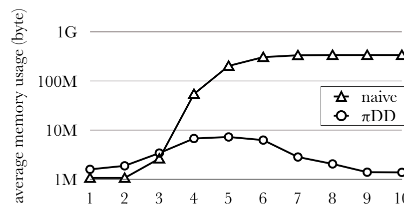


Figure 11: Memory consumption when $n = 10$.

Figure 10 and 11 show the results for classical patterns. These results show the average time and memory consumption for all pattern, where n is fixed at 10 and k is varied. Memory consumption of the naive method is proportional to the number of pattern-avoiding permutations, but that of the πDD method decreases when $k \geq 5$. 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, the computation time of the naive method is in proportion to $\binom{n}{k}$, which is the number of subsequences that must be checked.

Table 4 presents the results for vincular patterns. We generated Baxter permutations, because there are the huge number of vincular patterns for each k, which is $k! \cdot 2^{k-1}$. Baxter permutations are defined as avoiding the two vincular patterns, $\underline{31}\ \underline{42}$ and $\underline{24}\ \underline{13}$, and appear in many mathematical problems [1, 6, 7]. Our algorithm generates $Av_n(\underline{31}\ \underline{42}, \underline{24}\ \underline{13}) = S_n \setminus (C_n(\underline{31}\ \underline{42}) \cup C_n(\underline{24}\ \underline{13}))$. On the other hand, the naive method checks the order isomorphism between all k-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 and consumes less memory than the naive method.

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

The number of permutations and the size of the corresponding πDD are shown in Table 5, where it is clear that πDD achieves a high compression ratio.

Table 4: 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	0.036	2752
9	58202	0.028	4164	0.380	5676
10	326240	0.060	12984	4.888	37864
11	1882960	0.212	26828	65.556	181112
12	11140560	0.908	98924	941.331	1442904
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 5: the πDD for Baxter permutations ($n = 15$).

	#permutations	#nodes in πDD
S_n	1307674368000	105
$C_n(\underline{31}\ \underline{42}) \cup C_n(\underline{24}\ \underline{13})$	1305081026414	4094585
$Av_n(\underline{31}\ \underline{42}, \underline{24}\ \underline{13})$ = Baxter perm.	2593341586	2158472

Table 6: Experimental results for generating $Av_n\binom{\overline{12}3}{\underline{23}1}$.

n	$\#Av_n\binom{\overline{12}3}{\underline{23}1}$	πDD Method		Naive Method	
		time (sec)	memory (KB)	time (sec)	memory (KB)
8	5335	0.004	3084	0.024	3080
9	31240	0.004	3084	0.188	3972
10	201608	0.020	4196	2.260	20080
11	1422074	0.052	12948	30.638	156568
12	10886503	0.184	26436	444.364	1191712
13	89903100	0.916	98656	6962.859	9834280
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	—	—

We show the results for bivincular patterns. Specifically, we generated $Av_n\binom{\overline{12}3}{\underline{23}1}$, which is known to be related to chord diagrams, (2+2)-free posets and ascent sequences [5]. Table 6 presents the result of generating $Av_n\binom{\overline{12}3}{\underline{23}1}$. 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 $Av_n\binom{\overline{12}3}{\underline{23}1}$ are all decreasing. For $n = 13$, the πDD method can generate the permutations in under 1 s while the naive one requires about 2 h.

6 Conclusion

In this paper, we proposed an algorithm for generating several pattern-avoiding permutations using πDDs. Experimental results show that our algorithm is faster and consumes less memory than the naive method. In addition, our approach outputs a πDD on memory, which has rich set 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.

A future work is to improve further the computation time and memory consumption of our algorithm. 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 to populate a floorplan database.

References

[1] E. Ackerman, G. Barequet, and R. Pinter. A Bijection Between Permutations and Floorplans, and its Applications. *Discrete Applied Mathematics*, 154(12):1674–1684, 2006.

[2] E. Babson and E. Steingrı́msson. Generalized Permutation Patterns and a Classification of the Mahonian Statistics. *Séminaire Lotharingien de Combinatoire*, 44, 2000.

[3] M. 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.

[4] P. Bose, J. Buss, and A. Lubiw. Pattern matching for permutations. *Information Processing Letters*, 65(5):277–283, 1998.

[5] M. Bousquet-Mélou, A. Claesson, M. Dukes, and S. Kitaev. (2+2)-free posets, ascent sequences and pattern avoiding permutations. *J. Comb. Theory Ser. A*, 117(7):884–909, Oct. 2010.

[6] H. Canary. Aztec Diamonds and Baxter Permutations. *The Electronic Journal of Combinatorics*, 17, 2010.

[7] E. Fusy. Bijective Counting of Involutive Baxter Permutations. *Fundam. Inf.*, 117(1-4):179–188, Jan. 2012.

[8] L. Ibarra. Finding pattern matchings for permutations. *Information Processing Letters*, 61(6), 1997.

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

[10] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[11] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[12] D. Marinov and R. Radoičić. Counting 1324-avoiding permutations. *The Electronic Journal of Combinatorics*, 9(2), 2003.

[13] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. *Proc. of 30th ACM/IEEE Design Automation Conf.(DAC-93)*, pages 272–277, 1993.

[14] S. 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*, pages 90–104, 2011.

[15] V. R. Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. *Proc. of the fifth annual ACM symposium on Theory of computing*, pages 268–277, 1973.

[16] Z. E. Stankova. Forbidden subsequences. *Discrete Math.*, 132:291–316, 1994.

[17] H. Wilf. The patterns of permutations. *Discrete Math.*, 257:575–583, 2002.

[18] B. Yao, H. Chen, C. K. Cheng, and R. L. Graham. Floorplan representations: Complexity and connections. *ACM Transactions on Design Automation of Electronic Systems*, 8(1):55–80, 2003.