順列二分決定グラフを用いたパターン回避順列の列挙索引化

井上 祐馬[1,2,a)]　戸田 貴久[2,1]　湊 真一[1,2]

概要：パターン回避順列とは，任意の部分列が特定の順序関係を持たないような順列をいう．パターン回避順列は，VLSI の設計などの応用に現れるフロアプランとの対応が明らかになるなど，応用分野でも近年注目を集めている．本稿では，順列集合を効率的に扱うことができる πDD というデータ構造に基づき，パターン回避順列，およびその拡張である隣接条件付きパターン回避順列を列挙索引化するアルゴリズムを示す．

キーワード：列挙，πDD，パターン回避順列，隣接条件付きパターン回避順列

Enumerating and Indexing of Pattern Avoiding Permutations Using πDDs

Yuma Inoue[1,2,a)]　Takahisa Toda[2,1]　Shin-ichi Minato[1,2]

Abstract: Pattern avoiding permutation is a permutation whose subsequences don't match a certain order. Pattern avoiding permutation is discussed on not only theoretical areas but also application areas, for example, floorplan for compacting VLSI. In this paper, we provide the algorithms for enumerating and indexing of pattern avoiding permutations and vincular pattern avoiding permutations, which have a extend definition of pattern avoiding, based on the efficient data structure for the set of permutations, πDDs.

Keywords: enumerating, πDD, pattern avoiding permutation, vincular pattern avoiding permutation

1. はじめに

順列 $\boldsymbol{\pi}$ がパターン $\boldsymbol{\sigma}$ を回避するとは，$\boldsymbol{\pi}$ の任意の部分列とパターンとなる順列 $\boldsymbol{\sigma}$ とが同じ順序関係を持たないことをいう．与えられたパターンを回避する順列を，一般にパターン回避順列 (*Pattern Avoiding Permutation*) と呼ぶ [4]．

パターン回避順列はこれまで数え上げ組合せ論の分野で盛んに研究されており，多くの研究者たちの興味の対象は主に特定のパターンを回避する順列の個数やその個数に基づくクラス分類，同じ分類内での対応関係などの理論的研究であった [3], [10]．しかし，一部のパターン回避順列に関する研究結果として，VLSI 設計などの応用が考えられるフロアプランのいくつかのモデルとの間に一対一対応が存在することが近年明らかになり [1]，実用的応用分野にも研究の裾野が広がっている．例えばパターン回避順列を列挙し，索引化してデータベースとして保持しておくことは，フロアプランのデータベースを保持しているのと同等の意味を持つ．このデータベース上でさらにフロアプランの条件絞り込みやランダムサンプリングを行うことは，VLSI の最適化を行う上で有用と考えられる．このように，今までの理論的研究では注目されていなかったパターン回避順列の列挙も，今後は重要な課題として研究が進んでいくものと考えられる．

本稿ではこの流れに先駆け，これまでに効率的なアルゴリズムが考えられていないパターン回避順列の列挙索引化を行うアルゴリズムを提案する．これは，順列集合を効率的に圧縮，操作できるデータ構造 πDD[9] を用いることに

[1] 北海道大学 情報科学研究科
Hokkaido University IST
[2] JST ERATO 湊離散構造処理系プロジェクト
JST ERATO Project
[a)] yuma@mx-alg.ist.hokudai.ac.jp

ⓒ 2013 Information Processing Society of Japan

より，計算時間のみならずパターン回避順列を保持するためのメモリ空間も節約できるという特徴を持つ．また，パターン回避の定義を更に拡張した隣接条件付きパターン回避順列 (*Vincular Pattern Avoiding Permutation*)[2] についても，列挙索引化が行えるよう拡張したアルゴリズムを示す．

本稿の構成は以下の通りである．2 節において順列のパターン，パターン回避の概念を導入する．続く 3 節にて提案手法で用いる πDD について説明する．4 節でパターン回避順列，および，隣接条件付きパターン回避順列の列挙索引化のアルゴリズムを解説し，5 節でナイーブな列挙手法との計算機比較実験に基づいて，本手法の有効性について考察する．最後の 6 節で本稿のまとめと今後の課題について述べる．

2. パターン回避順列

2.1 順列と置換

順列とは要素を順番に意味を持たせて並べたものである．本稿では要素が過不足なくちょうど一度ずつだけ現れる順列のみを扱う．すなわち，順列を $\boldsymbol{\pi} = (\pi_1\ \pi_2\ \ldots\ \pi_n)$ と表記するとき，以下の 2 条件を満たす．

(1) $1 \leq i \leq n$ について，$\pi_i \in \{1, 2, \ldots, n\}$

(2) $i \neq j \Leftrightarrow \pi_i \neq \pi_j$

また，この順列は置換とみなせる．すなわち，順列 $\boldsymbol{\pi} = (\pi_1\ \pi_2\ \ldots\ \pi_n)$ は，$\boldsymbol{\pi}(i) = \pi_i$ で定義される全単射関数 $\boldsymbol{\pi}$ を表している，と捉えられる．このとき，$\boldsymbol{\pi}(i) = \pi_i$ を $i\boldsymbol{\pi} = \pi_i$ とも表記する．全ての i について $i\boldsymbol{\pi} = i$ を満たす $\boldsymbol{\pi}$ を恒等置換とよび，$\boldsymbol{e_n}$ と書く．置換 $\boldsymbol{a}, \boldsymbol{b}$ の合成 $\circ$ は，

$$\boldsymbol{a} \circ \boldsymbol{b} = ((1\boldsymbol{a})\boldsymbol{b}\ (2\boldsymbol{a})\boldsymbol{b}\ \ldots\ (n\boldsymbol{a})\boldsymbol{b}) = (b_{a_1}\ b_{a_2}\ \ldots\ b_{a_n})$$

と定義される．これを置換の積ともよぶ．置換の積は明らかに可換ではない．

2 つの要素を除き恒等写像となるような置換は特別に互換と呼ばれる．2 つの要素が i と j であるとき，この互換を $\tau_{i,j}$ と表記する．任意の置換は互換のみからなる積で表せることが知られており，さらに，互換の積の順序に規則性を持たせることで，$n-1$ 個以下の一意な互換の積形となることが示されている [9]．図 **1** はその一例である．

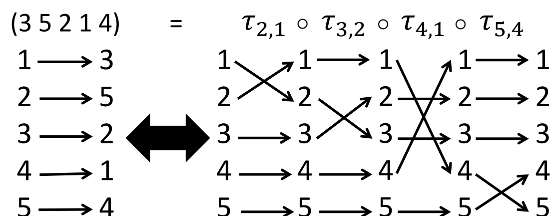

図 **1** 置換と等価な互換の積
Fig. 1 Permutation and corresponding transpositions

2.2 順列のパターンとパターン回避順列

$\boldsymbol{\sigma}$ と同じ順序関係をもつ長さ k の部分列が $\boldsymbol{\pi}$ に存在するとき，$\boldsymbol{\pi}$ は $\boldsymbol{\sigma}$ をパターンとして含むという．より厳密には，順列 $\boldsymbol{\pi} = (\pi_1\ \pi_2\ \ldots\ \pi_n)$ がパターン $\boldsymbol{\sigma} = (\sigma_1\ \sigma_2\ \ldots\ \sigma_k)$ を含むとは，すべての x, y に対し，$\pi_{i_x} < \pi_{i_y}$ のとき，かつそのときに限り必ず $\sigma_x < \sigma_y$ を満たすような $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ が存在することをいう [4]．

例えば，$\boldsymbol{\pi} = (2\ 5\ 3\ 1\ 4)$，$\boldsymbol{\sigma} = (3\ 1\ 2)$ とすると，$\boldsymbol{\pi}$ の部分列 (5 1 4) は (3 1 2) と同じ順序関係をもっているため，$\boldsymbol{\pi}$ は $\boldsymbol{\sigma}$ をパターンとして含んでいる．

順列 $\boldsymbol{\pi}$ がパターン $\boldsymbol{\sigma}$ を含むとき，$\boldsymbol{\pi}$ を $\boldsymbol{\sigma}$-パターン含有順列と呼ぶ．逆に，順列 $\boldsymbol{\pi}$ がパターン $\boldsymbol{\sigma}$ を含まないとき，$\boldsymbol{\sigma}$-パターン回避順列，あるいは単に $\boldsymbol{\sigma}$-回避 ($\boldsymbol{\sigma}$-*Avoiding*) であるという．

2.3 隣接条件付きパターン回避

パターン回避順列の一般化のひとつに，隣接条件付きパターン回避順列 (*Vincular Pattern Avoiding Permutation*) がある [2]．これは数字の順序関係だけではなく，隣接関係も考慮に入れたパターン回避順列である．隣接している必要がない場合には数字間をハイフンで区切り，隣接しなければならない場合には区切り記号を用いずに表記する．通常のパターン回避と違い，ハイフンが間にないときは隣接する必要があることに注意しなければならない．

例えば，$\boldsymbol{\pi} = (2\ 5\ 3\ 1\ 4)$，$\boldsymbol{\sigma} = (3\ 1\text{-}2)$ のときを考える．$\boldsymbol{\pi}$ の部分列 (5 1 4) は，順序関係は等しいが 5 と 1 とが隣接していないためパターンに一致しない．しかし，(5 3 4) は 5 と 3 が隣接しており，順序関係も等しいため，パターンに一致している．故に，$\boldsymbol{\pi}$ は隣接条件付きパターン $\boldsymbol{\sigma}$ を含んでいる．

2.4 関連研究

パターン回避順列が初めて現れたのは Knuth の提案したスタックソートである [7]．スタックソートでは，操作として 1 つスタックへの出し入れのみを利用してソートを行う．スタックソートにより整列可能な順列は，(2 3 1)-回避であることが知られている．また，スタックを 2 回利用できる場合など様々な類題が考えられている [6] が，これらの整列可能性も特定のパターンを回避しているか否かで判定できる．

パターン回避順列の列挙に関して，Wilf はパターンが恒等置換である場合の列挙アルゴリズムを文献 [11] にて示した．同時に，一般のパターン回避について，どれほど効率的に列挙が可能かという問題を提示した．Bose らは一般のパターン回避順列の判定問題が NP 完全であることを示すとともに，パターンがある制約を満たす場合に限り有効な $O(kn^6)$ の判定アルゴリズムを示した [4]．後にこれは Ibarra により $O(kn^4)$ に改善されている [5]．しかし，一般

のパターン回避順列の列挙アルゴリズムに関する先行研究は筆者が知る限り存在しない．

応用可能性を示した研究結果として，VLSI 設計などに用いられるフロアプランとパターン回避順列との対応がある．矩形をいくつかの矩形に分割し，ブロックを割り当てる問題がフロアプランである．このうち，分割線の交差点が T 字のみであるモザイクフロアプランが Baxter permutation と対応し，再帰的な分割により表現できるスライシングフロアプランが separable permutaion と対応していることが示されている [1], [12]．Baxter permutation は (3-1 4-2)-回避かつ (2-4 1-3)-回避の順列であり，separable permutation は (3 1 4 2)-回避かつ (2 4 1 3)-回避の順列である．

3. πDD

本章で扱う πDD は ZDD と呼ばれるデータ構造を基盤にしている．まずは ZDD について説明する．

3.1 ZDD

ZDD(*Zero-suppressed Binary Decision Diagram*) は組合せ集合を効率的に扱うデータ構造である [8]．多数の異なるアイテムの中からいくつかを選んで集めたものを組合せとよぶ．組合せ集合とは，この組合せを要素として集めた集合である．すなわち，組合せ集合は集合の族であるとも言える．

組合せ集合を二分木で表現することを考える．それぞれの節点にはアイテムを表すラベルがついている．アイテムが組合せに含まれるか否かを枝分かれで表現し，根から葉までのパスが 1 つの組合せを表す．アイテムが組合せに含まれることを示す枝を 1-枝，含まれないことを示す枝を 0-枝と呼ぶ．また，辿ってきたパスが表す組合せが集合に含まれることを表す葉を 1-終端節点，含まれないことを表す葉を 0-終端節点と呼ぶ．

ZDD とは，以下の 2 つの規則に基づいてこの二分木を圧縮したものである (図 **2**)．

等価な節点の共有　同じアイテムを表す 2 つの節点について，1-枝が指す節点同士，0-枝が指す節点同士がそれぞれ等しいとき，これを 1 つの節点にまとめる．

冗長な節点の削除　1-枝が 0-終端節点を指す節点を削除し，この節点を指していた枝をこの節点の 0-枝が指す先に付けかえる．

圧縮規則に基づき，二分木を ZDD に圧縮する様子を図 **3** に示す．

このような圧縮を行っても，ZDD サイズ，すなわち節点数はアイテム数の指数オーダーとなることがある．しかし，類似する組合せが多数存在するときや，全体のアイテム数に対し各組合せに現れるアイテム数が少ない場合などに大きな圧縮効果が得られることが実験的に示されている [8]．

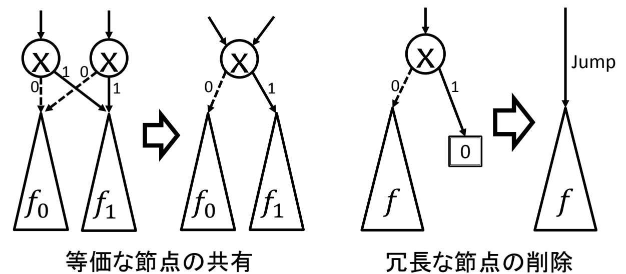

図 **2**　2 つの圧縮規則
Fig. 2　Two reduction rules on ZDDs

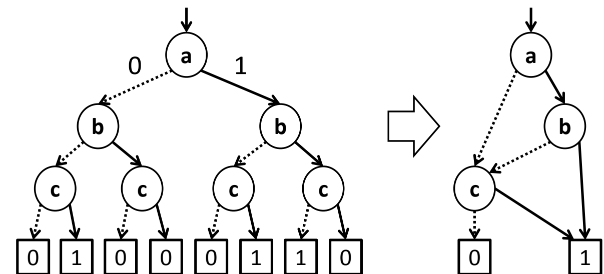

図 **3**　組合せ集合を表す二分木と ZDD
Fig. 3　Binary tree and ZDD for the set of combinations

また，ZDD の大きな特徴として，圧縮された状態のまま様々な集合演算が可能である点が挙げられる．例えば，和集合，積集合，差集合などの二項集合演算を 2 つの ZDD サイズの積に比例した時間で計算できる．

3.2 πDD

πDD [7] は ZDD を基にして大きさが n である順列の集合をコンパクトに表すデータ構造である．順列集合の各順列を，2.1 節で述べた規則を用いて一意な互換の積で表す．このとき，互換の積には同じ互換は 2 個以上現れないので，互換の積を互換の組合せとみなすことができる．順列集合を互換の組合せ集合とみなし，これを ZDD で表現したものが πDD である．

πDD では，1-終端節点から根へ向かう方向に互換の列が並ぶように節点の順序付けを行う．順列集合とそれに対応する πDD の一例を図 **4** に示す．

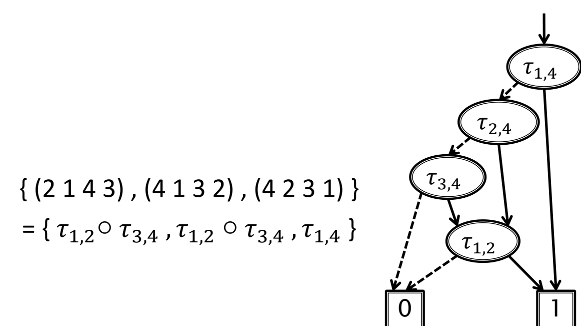

図 **4**　順列集合とそれを表す πDD
Fig. 4　Permutation set and its πDD Representation

πDD は ZDD と同様に，和集合，積集合，差集合，直積などの様々な集合演算を圧縮した状態のまま適用できる．ここで 2 つの順列集合 P, Q の直積演算 $\times$ とは，P と Q に含まれる順列を一つずつ取り出した全ての組における，置換の合成の和集合と定義する．ゆえに，$P \times Q = \{\boldsymbol{x} \circ \boldsymbol{y} \mid x \in P, y \in Q\}$ と書ける．例えば，$P = \{(2\ 1\ 3\ 4), (1\ 3\ 2\ 4)\}, Q = \{(4\ 1\ 3\ 2), (2\ 3\ 4\ 1)\}$ なら，

$$\begin{aligned}
&P \times Q \\
&= \{(2\ 1\ 3\ 4), (1\ 3\ 2\ 4)\} \times \{(4\ 1\ 3\ 2), (2\ 3\ 4\ 1)\} \\
&= \{(2\ 1\ 3\ 4) \circ (4\ 1\ 3\ 2), (2\ 1\ 3\ 4) \circ (2\ 3\ 4\ 1), \\
&\qquad (1\ 3\ 2\ 4) \circ (4\ 1\ 3\ 2), (1\ 3\ 2\ 4) \circ (2\ 3\ 4\ 1)\} \\
&= \{(1\ 4\ 3\ 2), (3\ 2\ 4\ 1), (4\ 3\ 1\ 2), (2\ 4\ 3\ 1)\}
\end{aligned}$$

である．置換の合成が可換でないことからもわかる通り，直積演算も可換ではない．

以下の定理 1 より，順列集合に含まれるすべての順列の並べ替えを 1 回の直積演算により一括で行える．

定理 1　任意の順列集合 P と，$\boldsymbol{a} = (a_1\ a_2\ \ldots\ a_n)$ のみからなる順列集合 A を考える．直積 $A \times P$ は，P に含まれる全ての順列を $(a_1\ a_2\ \ldots\ a_n)$ の順に並べ替えた順列集合に等しい．

証明

$$\begin{aligned}
A \times P &= \{\boldsymbol{a} \circ \boldsymbol{p} \mid \boldsymbol{p} = (p_1\ p_2\ \ldots\ p_n) \in P\} \\
&= \{(p_{a_1}\ p_{a_2}\ \ldots\ p_{a_n}) \mid \boldsymbol{p} = (p_1\ p_2\ \ldots\ p_n) \in P\}
\end{aligned}$$

より明らか．∎

定理 1 を用いると，例えば $\boldsymbol{a} = (n\ n-1\ \ldots\ 2\ 1)$ のとき，$A \times P$ は P に含まれる順列が全て逆順になった順列集合を表す．

定理 1 は簡単のため，A に含まれる順列を 1 つとしたが，複数の順列が含まれる場合，直積演算の結果は全ての並び替えの和集合となる．定理 1 より，直積演算は πDD を用いた順列集合の操作において重要な演算といえるが，他の演算に比べ計算量が大きいというデメリットもある [9].

4. パターン回避順列の列挙索引化アルゴリズム

パターン回避順列および隣接条件付きパターン回避順列の列挙索引化を実現するアルゴリズムを説明する．以降，列挙する順列の長さを n，パターンの長さを k とする．

4.1 $\boldsymbol{\sigma}$-パターン回避順列の列挙索引化

$\boldsymbol{\sigma}$-パターン回避順列を列挙するためには，$\boldsymbol{\sigma}$-パターン含有順列を列挙し全体集合と差集合をとればよい．

全体集合は **Algorithm**1 で列挙可能である．全体集合を表す πDD のサイズは $O(n^2)$ で，要素数 $n!$ に比べて大幅に圧縮されている．このように，通常のデータ構造では計算時間やメモリの都合上扱いづらい差集合演算や全体集合を πDD を用いることで効果的に利用している点も，本手法の特徴のひとつである．

Algorithm 1 全体集合を表す πDD の計算

```
πDD U ← {e_n}
for i = 1 to n do
  for j = 1 to i − 1 do
    U ← U ∪ (U × {τ_{i,j}})
  end for
end for
```

以下より $\boldsymbol{\sigma}$-パターン含有順列を列挙する方法を考える．

4.2 $\boldsymbol{\sigma}$-パターン含有順列の列挙索引化

ここで一度，順列 $\boldsymbol{\pi}$ にパターン $\boldsymbol{\sigma}$ が含まれているか判定する方法について考える．

長さが k である $\boldsymbol{\pi}$ の部分列全体の集合を P_k とする．$\boldsymbol{\pi}$ に $\boldsymbol{\sigma}$ が含まれるならば，$\boldsymbol{\sigma}$ と同じ順序関係を持つような $\boldsymbol{p} \in P_k$ が少なくとも 1 つ存在する．よって，$\boldsymbol{\pi}$ に $\boldsymbol{\sigma}$ が含まれるか判定するナイーブな方法として，$\boldsymbol{\pi}$ の長さ k の部分列を全列挙し，$\boldsymbol{\sigma}$ の順序関係と比較する方法が考えられる．$\boldsymbol{p}$ と $\boldsymbol{\sigma}$ が同じ順序関係を持つとき，$\boldsymbol{p}$ は $\{1, 2, \ldots, n\}$ の n 個から k 個を選ぶ組合せを，$\boldsymbol{\sigma}$ に従った順序に並べ替えたものとみなすことができる．ゆえに，以下の手順によってパターン含有順列の列挙が可能であることがわかる．

(1) n 個から k 個選ぶ組合せを前方 k 個に昇順に集めた順列を作り，それぞれに対し，残った後方 $n-k$ 個の要素について全ての順序を試し，和集合を作る．

(2) (1) の集合に含まれる全順列の前方 k 個をパターン $\boldsymbol{\sigma}$ に一致するように並び替える．

(3) (2) の集合の前方 k 個の要素を，順序を変えずに n 個の位置から k 個選んだ新しい位置に割り当てる．可能な位置の選び方すべてに対しこれを行う．後方 $n-k$ 個の要素の新しい位置は問わないが，1 つの割り当て方に対して一意に定める必要がある．

ここで，(1) から (3) への手順はすべて並べ替えと捉えることができる．すなわち，定理 1 により，πDD の直積演算を用いることにより実現できる．具体的には，以下の 3 つの順列集合，

comb　n 個から k 個選ぶ組合せを前方 k 個に集め，かつ，後方 $n-k$ 個の全ての並びを列挙した順列集合

pattern　整列された順列の前方 k 個をパターン $\boldsymbol{\sigma}$ に一致するように並べ替えた順列のみからなる順列集合

pos　n 個の位置から k 個の位置を選ぶ選び方それぞれについて，選んだ位置に昇順に 1 から k までが割り当てられているような順列がちょうど 1 つ現れる順列集合

を πDD で表し，**pos** $\times$ **pattern** $\times$ **comb** を計算することにより，パターン含有順列が得られる．ただし，この手順は重複数え上げが起こりうるため，集合として扱わなければ同じ順列が複数列挙される可能性がある点に注意しなければならない．

pattern は $\boldsymbol{\sigma}$ を規則に従って互換の積に変換しながら πDD を構築することで簡単に得られる．以降の節では，**pos** と **comb** を生成する方法について述べる．

4.3 pos を生成するアルゴリズム

$1 \le i < k$ ついて $p_i < p_{i+1}$ を満たす k 個の位置 $p_1, p_2, ..., p_k$ に，それぞれ 1 から k を割り当てることを考える．このとき，全ての $1 \le i \le k$ について τ_{i,p_i} をかければよいように思える．しかし例えば $p_1 = 2, p_2 = 3$ のとき，$\tau_{2,3}$ をかけてから $\tau_{1,2}$ をかけると，$\tau_{2,3} \circ \tau_{1,2} = (2\ 3\ 1\ 4\ \ldots\ n)$ となり，$\pi_2 \neq 1, \pi_3 \neq 2$ となってしまう．一般に，i 番目にかける互換を $\tau_{k_i,p_{k_i}}$ とすると，$k_j = p_{k_i}$ となるような $j < i$ が存在するとき，このような問題が生じる．この問題は，i の昇順に τ_{i,p_i} をかけることで解決できる．先の例も，$\tau_{1,2}$ をかけてから $\tau_{2,3}$ をかけると，$\tau_{1,2} \circ \tau_{2,3} = (3\ 1\ 2\ 4\ \ldots\ n)$ となり正しく割り当てられていることがわかる．

全ての位置の選び方についてこれを行い，最後に和集合をとることで **pos** は列挙可能である．しかし，これは全ての位置の選び方の数 $O(n^k)$ に比例する回数の計算が必要となってしまうため効率が悪い．

ところで，p 個から k 個を選ぶ選び方が何通りあるか数え上げる方法として，動的計画法が考えられる．数のみを求める場合，$D_{i,j}$ を「i 個を j 以下の数から選んだときの選び方の数」とおくと，以下の漸化式が成り立つ．

$$D_{0,0} = 1$$
$$D_{i,j} = \begin{cases} D_{i-1,j-1} + D_{i,j-1} & (1 \le i \le k, 1 \le j \le n) \\ 0 & (otherwise) \end{cases}$$

p 個から k 個を選ぶ選び方の数は，$D_{k,n}$ を求めればよい．

これを集合に拡張し，$D_{i,j}$ を「i 個を j 以下の数から選んだときの選び方の集合」とおく．選び方を順列の前方 k 個の要素で表すと，i 個目を j にするときは $\tau_{i,j}$ をかければよいので，以下の漸化式が成り立つ．

$$D_{0,0} = \{e_n\}$$
$$D_{i,j} = \begin{cases} (D_{i-1,j-1} \times \{\tau_{i,j}\}) \cup D_{i,j-1} \\ \qquad\qquad (1 \le i \le k, 1 \le j \le n) \\ \phi \qquad\qquad (otherwise) \end{cases}$$

このとき，各 $D_{i,j}$ において互換がかけられる順は昇順である．また，各順列集合に含まれる全ての順列の前方 k 個が昇順であることも保証される．よって $\mathbf{pos} = D_{k,n}$ であり，求めるアルゴリズムは **Algorithm**2 のようになる．

Algorithm 2 pos を表す πDD の計算
```
πDD D_{0,0} ← {e_n}
for i = 0 to k do
  for j = 1 to n do
    if i > 0 then
      D_{i,j} ← (D_{i−1,j−1} × {τ_{i,j}}) ∪ D_{i,j−1}
    else
      D_{i,j} ← D_{i,j−1}
    end if
  end for
end for
πDD pos ← D_{k,n}
```

4.4 comb を生成するアルゴリズム

comb も **pos** と同様に，動的計画法を用いて生成することができる．しかし，**comb** では昇順に τ_{i,p_i} をかけてしまうとうまくいかない．**comb** では τ_{i,p_i} をかける順を降順にすればよい．また，**comb** の場合，後方 $n-k$ 個の全ての順を列挙する必要があるため，最後にこの並べ替えも行う．**comb** を生成するアルゴリズムを **Algorithm**3 に示す．

Algorithm 3 comb を表す πDD の計算
```
πDD D_{0,0} ← {e_n}
for i = 0 to k do
  for j = 1 to n do
    if i > 0 then
      D_{i,j} ← (D_{i−1,j−1} × {τ_{k+1−i,n+1−j}}) ∪ D_{i,j−1}
    else
      D_{i,j} ← D_{i,j−1}
    end if
  end for
end for
πDD suf ← {e_n}
for i = k + 1 to n do
  for j = k + 1 to i − 1 do
    suf ← suf ∪ (suf × {τ_{i,j}})
  end for
end for
πDD comb ← suf × D_{k,n}
```

4.5 $\boldsymbol{\sigma}$-隣接条件付きパターン回避順列の列挙索引化

隣接条件付きパターン回避順列の列挙についても同様に隣接条件付きパターン含有順列を列挙して全体集合から差集合をとることで得られる．隣接条件付きパターン含有順列を得る方法もパターン含有順列とほぼ同様だが，位置の制約，すなわち **pos** のみが異なる．これを **pos'** と呼ぶこととする．$\boldsymbol{\sigma}$ によって隣接条件が付けられている箇所がパターンの x 番目と $x+1$ 番目の間だとすると，**pos'** に含まれるのは，**pos** に含まれる順列のうち，選ばれた位置の x 番目と $x+1$ 番目が隣り合っているものということになる．これは前述の動的計画法の $i = x$ において，$D_{i,j}$ に $D_{i,j-1}$ を加えないことに相当する．よって，以下の **Algorithm**4 のように修正することで，**pos'** の生成が可能である．

表 1　パターン回避順列列挙の実行時間

Table 1　Run time for enumerating pattern avoiding permutations.

n		提案手法 k				ナイーブ法 k			
		2	3	4	5	2	3	4	5
8	最良 (sec)	0.000(12)	0.000(213)	0.000(1423)	0.000(12543)	0.004(21)	0.012(132)	0.044(1243)	0.072(12354)
	平均 (sec)	0.000	0.002	0.004	0.001	0.008	0.013	0.052	0.075
	最悪 (sec)	0.000(12)	0.008(123)	0.012(1234)	0.008(12345)	0.012(12)	0.020(123)	0.072(4132)	0.088(12543)
9	最良 (sec)	0.000(21)	0.000(321)	0.004(4132)	0.004(12354)	0.056(21)	0.104(213)	0.608(3241)	1.424(23145)
	平均 (sec)	0.004	0.005	0.009	0.009	0.058	0.108	0.641	1.469
	最悪 (sec)	0.008(12)	0.016(123)	0.016(1234)	0.024(31425)	0.060(12)	0.112(123)	0.676(4123)	1.512(15243)
10	最良 (sec)	0.004(21)	0.004(321)	0.016(3421)	0.024(12534)	0.464(21)	1.096(231)	8.584(2314)	30.566(42513)
	平均 (sec)	0.008	0.011	0.028	0.036	0.468	1.106	8.907	31.241
	最悪 (sec)	0.012(12)	0.020(123)	0.044(2134)	0.060(13425)	0.472(12)	1.112(312)	9.201(1432)	31.794(54231)
11	最良 (sec)	0.000(21)	0.012(321)	0.044(4321)	0.092(45321)	5.268(12)	12.517(231)	122.304(3241)	595.009(42513)
	平均 (sec)	0.008	0.029	0.101	0.174	5.292	12.608	126.194	609.160
	最悪 (sec)	0.016(12)	0.052(123)	0.152(1234)	0.276(24513)	5.316(21)	12.697(312)	129.668(1324)	622.387(12435)
12	最良 (sec)	0.004(21)	0.024(321)	0.152(4321)	0.428(54321)	—	—	—	—
	平均 (sec)	0.014	0.087	0.444	0.921	—	—	—	—
	最悪 (sec)	0.024(12)	0.156(123)	0.780(1324)	1.392(13425)	—	—	—	—
13	最良 (sec)	0.016(21)	0.048(321)	0.568(4312)	1.824(54321)	—	—	—	—
	平均 (sec)	0.022	0.284	1.787	4.309	—	—	—	—
	最悪 (sec)	0.022(12)	0.544(123)	3.072(1324)	6.888(14325)	—	—	—	—
14	最良 (sec)	0.008(21)	0.116(321)	1.960(4321)	6.448(54321)	—	—	—	—
	平均 (sec)	0.032	1.029	6.769	19.036	—	—	—	—
	最悪 (sec)	0.056(12)	1.968(123)	12.021(1324)	32.370(14325)	—	—	—	—
15	最良 (sec)	0.016(21)	0.300(321)	5.688(4321)	23.814(54321)	—	—	—	—
	平均 (sec)	0.052	3.513	24.771	85.655	—	—	—	—
	最悪 (sec)	0.088(12)	6.860(123)	48.415(1324)	160.562(14325)	—	—	—	—

Algorithm 4 pos' を表す πDD の計算

πDD $D_{0,0} \leftarrow \{e_n\}$
for $i = 0$ to k **do**
　for $j = 1$ to n **do**
　　if $i > 0$ **then**
　　　if $i - 1$ 番目と i 番目が隣接しなければならない **then**
　　　　$D_{i,j} \leftarrow (D_{i-1,j-1} \times \{\sigma_{i,j}\})$
　　　else
　　　　$D_{i,j} \leftarrow (D_{i-1,j-1} \times \{\sigma i, j\}) \cup D_{i,j-1}$
　　　end if
　　else
　　　$D_{i,j} \leftarrow D_{i,j-1}$
　　end if
　end for
end for
πDD **pos'** $\leftarrow D_{k,n}$

これを利用し，全体集合から **pos'** $\times$ **pattern** $\times$ **comb** の差集合をとることで，隣接条件付きパターン回避順列の列挙索引化が可能となる．

5. 実験結果

提案手法を C++言語を用いて実装し，実験を行った．また，ナイーブ法として長さ n の順列を全列挙し，長さ k の全ての部分列の順序関係を比較するアルゴリズムも C++言語で実装し，同様の条件で実験を行った．実験環境として，Intel® Core™ i7-3930K CPU 3.20GHz 6 Core，Ubuntu 12.04.1 LTS 64bit OS，64GB メモリの計算機を用いた．CPU は複数コアを有しているが，今回のプログラムでは並列化は行っていない．

始めにパターン回避順列の列挙に関する実験結果を示す．この実験では $2 \leq k \leq 5$ のパターンを全て生成し，それぞれについてパターン回避順列の集合を求めた．その際の実行時間を**表 1**，使用メモリ量を**表 2** に示す．同じパターン長 k のうち，もっとも高速，もしくはメモリ使用量が小さい結果を最良，もっとも低速，もしくはメモリ使用量が大きい結果を最悪とし，括弧内はその結果を出した具体的なパターンを示している．また，平均は同じ k における全結果の平均を示している．

全体として，提案手法はナイーブ法を大きく上回っている．提案手法の最悪値とナイーブ法の最良値を比較しても，提案手法の方が効率的といえる．$n = 11, k = 5$ の平均値に注目すると，実行時間が約 3000 倍，メモリ使用量が約 100 倍の効率向上が確認できる．特に n が大きくなるほど顕著に差が出ているが，これは提案手法もナイーブ法同様に実行時間・メモリ使用量が指数的に増えていくものの，倍率が小さいことを示している．一方で，k の増加に伴う倍率は，ナイーブ法と提案手法の間で顕著な差は現れなかった．

また，最良と最悪の比に注目すると，ナイーブ法は最良

表 2　パターン回避順列列挙の使用メモリ

Table 2　Memory usage for enumerating pattern avoiding permutations.

n		提案手法				ナイーブ法			
		k				k			
		2	3	4	5	2	3	4	5
8	最良 (KB)	1600(12)	1596(123)	1596(4312)	1596(14532)	1072(12)	1332(231)	2152(1243)	5748(13524)
	平均 (KB)	1600.0	1598.7	1916.3	1599.4	1072.0	1335.3	2155.3	5751.5
	最悪 (KB)	1600(12)	1600(132)	1972(3412)	1600(12345)	1072(12)	1336(123)	2156(1234)	5752(12345)
9	最良 (KB)	1600(12)	1964(321)	2748(1243)	2744(14352)	1068(12)	1596(231)	10308(1342)	19452(13245)
	平均 (KB)	1600.0	2233.3	2933.5	2768.4	1070.0	1599.3	10312.0	19455.3
	最悪 (KB)	1600(21)	2768(123)	4208(2314)	2792(12453)	1072(21)	1600(123)	10316(2134)	19452(12345)
10	最良 (KB)	1600(21)	2760(312)	4212(4321)	7156(43521)	1072(12)	3472(123)	74676(3241)	295932(14532)
	平均 (KB)	1784.0	3509.3	6856.7	7376.7	1072.0	3472.0	74679.8	295935.5
	最悪 (KB)	1968(12)	4260(123)	7444(2134)	7700(13425)	1072(12)	3472(123)	74680(1234)	295940(14532)
11	最良 (KB)	1600(21)	4208(312)	7676(4321)	13720(54321)	1072(12)	6788(213)	361436(1324)	2884756(15423)
	平均 (KB)	2186.0	5876.7	17641.3	25233.3	1074.0	6791.3	361439.7	2884759.6
	最悪 (KB)	2772(12)	6792(123)	25084(1324)	27000(13425)	1076(21)	12.697(312)	361440(1324)	2884764(34215)
12	最良 (KB)	1976(21)	7112(321)	25316(4321)	49108(54321)	—	—	—	—
	平均 (KB)	3112.0	16106.7	48405.8	93525.2	—	—	—	—
	最悪 (KB)	4248(21)	25084(123)	94788(1324)	102940(14325)	—	—	—	—-
13	最良 (KB)	2764(21)	12708(321)	51436(4321)	188416(54321)	—	—	—	—
	平均 (KB)	4954.0	32130.7	168848.2	337123.4	—	—	—	—
	最悪 (KB)	7144(12)	51084(123)	201264(1324)	408460(35124)	—	—	—	—
14	最良 (KB)	2760(21)	24440(321)	187372(4321)	396432(54321)	—	—	—	—
	平均 (KB)	7754.0	111634.7	542621.2	1282547.8	—	—	—	—
	最悪 (KB)	12748(12)	190460(123)	779640(1324)	1587600(14325)	—	—	—	—
15	最良 (KB)	4228(21)	47620(321)	389140(4321)	1543108(54321)	—	—	—	—
	平均 (KB)	9112.0	234836.0	1560720.0	4986352.2	—	—	—	—
	最悪 (KB)	13996(12)	406440(123)	3092572(1324)	6471388(14325)	—	—	—	—

と最悪の差が現れていないのに対し，提案手法は最良と最悪に大きな差が生まれている．これは提案手法の実行時間や使用メモリが πDD の節点数に大きく依存するためであり，パターン回避順列の集合を πDD として表現したときに圧縮が効きやすいパターンと効きにくいパターンがあることが伺える．実際 $12 \leq n \leq 15$ では，最悪，最良のパターンはほぼ一致している．

次に，隣接条件付きパターン回避順列の列挙に関する実験結果を**表 3** に示す．隣接条件付きパターンはパターンの種類が多く，全て試すことは困難であるため，一例として Baxter permutation の列挙を行った．Baxter permutation は 2 つの隣接条件付きパターンを同時に回避している．ナイーブ法では順序関係の比較時に 2 つのパターンとの比較を同時に行った．提案手法では，2 つの隣接条件付きパターンを含有する順列集合をそれぞれ用意したあと，それらの和集合をとることで 2 つの隣接条件付きパターンのうちどちらかを含有する順列を列挙し，最後に全体集合から差集合をとることで列挙を行った．$B(n)$ は Baxter permutation の数を表している．

表 3　Baxter permutation 列挙の実験結果

Table 3　Experimental results for enumerating Baxter permutations.

n	$B(n)$	提案手法		ナイーブ法	
		時間 (sec)	メモリ (KB)	時間	メモリ
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	—	—

隣接条件付きパターン回避順列でも通常のパターン回避順列と同様，提案手法がナイーブ法を上回る結果が得られた．

$n = 15$ において，差集合計算のみの実行時間は 1.110sec であり，$n = 15$ の計算全体の 2%程度であった．計算時間のほとんどは 3 つの πDD の直積を計算する部分が占めており，46.020sec を要した．また，πDD のサイズは**表 4** に示す通り，高い圧縮効果が示されている．特に全体集合の

表 4　n=15 の Baxter permutation 列挙における πDD

Table 4　πDD for enumerating Baxter permutations(n=15).

	要素数	節点数
全体集合	1307674368000	105
パターン含有順列	1305081026414	4094585
パターン回避順列	2593341586	2158472

ように，互換の組合せとして類似した順列が多く含まれる集合では驚異的な圧縮率となることがある．さらに，パターン回避順列に対するパターン含有順列の比は，要素数については約 600 倍であるが πDD の節点数は 2 倍程度であり，大きな差はない．これは，通常のデータ構造では差集合演算を利用して補集合を求める手法が非効率的であるときでも，πDD ではほとんどオーバーヘッドなしに利用できる場合があることを示している．

$n \geq 18$ の列挙は実験機では計算に必要なメモリが足りないため，断念した．メモリを最も大きく使っているのは計算途中であり，最終的に得られる πDD のみの使用メモリは更に小さい値であるため，計算過程の工夫によりさらなる省メモリ化の可能性も考えられる．

6. おわりに

本稿では，パターン回避順列，および，その一般化である隣接条件付きパターン回避順列を πDD を用いて列挙索引化するアルゴリズムを提案した．また，実験結果より単純なアルゴリズムに比べ高速，省メモリであることがわかった．

今後の課題として，より省メモリな計算が可能となるようにアルゴリズムを改良することが挙げられる．また，実用的な応用の検討として特にフロアプランへの応用を見据え，例えばフロアプランの条件絞り込みなどの操作を πDD の演算で行えるよう調査を進めている．

参考文献

[1] Ackerman, E., Barequet, G. and Pinter, R.: A Bijection Between Permutations and Floorplans, and its Applications, *Discrete Applied Mathematics*, Vol. 154, No. 12, pp. 1674–1684 (2006).

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

[3] Bóna, M.: Exact enumeration of 1342-avoiding permutations: a close link with labeled trees and planar maps, *Journal of Combinatorial Theory*, Vol. 80, No. 2, pp. 257–272 (1997).

[4] Bose, P., Buss, J. and Lubiw, A.: Pattern matching for permutations, *Information Processing Letters*, Vol. 65, No. 5, pp. 277–283 (1998).

[5] Ibarra, L.: Finding pattern matchings for permutations, *Information Processing Letters*, Vol. 61, No. 6 (1997).

[6] J.West: Sorting twice through a stack, *Theoretical Computer Science*, Vol. 117, pp. 303–313 (1993).

[7] Knuth, D.: *The art of computer programming*, Vol. 1, Addison-Wesley Publishing Co., Reading MA (1973).

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

[9] Minato, S.: πDDs: A New Decision Diagram for Efficient Problem Solving in Permutation Space, *Proc. of 14th International Conference on Theory and Applications of Satisfiability Testing*, pp. 90–104 (2011).

[10] Stankova, Z. E.: Forbidden subsequences, *Discrete Math.*, Vol. 132, pp. 291–316 (1994).

[11] Wilf, H.: Ther patterns of permutations, *Discrete Math.*, Vol. 257, pp. 575–583 (2002).

[12] Yao, B., Chen, H., Cheng, C. K. and Graham, R. L.: Floorplan representations: Complexity and connections, *ACM Transactions on Design Automation of Electronic Systems*, Vol. 8, No. 1, pp. 55–80 (2003).

ⓒ 2013 Information Processing Society of Japan