あみだくじ数え上げ問題に対する高速解法

田中　勇真

近年 OR 分野全般において，列挙や数え上げ問題に対する注目が集まっている．列挙や数え上げ問題は入力されたパラメータ値の増加に対して，急速に解を求める時間が増加することが多い．そのため，アルゴリズムをうまく設計する必要があるが，コンピュータのリソースを限界近くまで利用するためアルゴリズムをどのように実装するかというのも非常に重要となる．本稿では，あみだくじ数え上げ問題に対する世界記録を更新した際に，どのようにアルゴリズムを考え，どのように実装したかについての経緯を簡単に紹介する．

キーワード：アルゴリズム，数え上げ，ソーティングネットワーク，あみだくじ

1. はじめに

本稿では，筆者があみだくじ数え上げ問題に対する世界記録を更新した際に，どのようにアルゴリズムを考え，どのように実装したかについて述べる．この結果は本年 7 月にバルセロナで開催された IFORS2014 において報告した（田中勇真，池上敦子，松井泰子，藤澤克樹，安井雄一郎 [1]）．日本人にとって，あみだくじは何かの割り当て方を運任せで決定する方法として非常に身近なものであり，読者の方々もよくご存知だと思う．しかしながら，世界的にみれば一般的なものではない．あみだくじと非常に関連深いものとして，ソーティング・ネットワーク (sorting network) がある．ソーティングネットワークとは，図 1 のように n 本の横線とそれらの 2 本の横線を結ぶいくつかの比較器（縦線）から構成され，入力側（左側）で与えられた任意の数列 $(x_1, x_2, \ldots, x_n)$ を，整列された数列 $(x'_1, x'_2, \ldots, x'_n), x'_1 \leq x'_2 \leq \cdots \leq x'_n$ にして出力（右側）するネットワークのことである．比較器は図 2 のように接続された 2 本の横線の値の大小を比較し，小さいほうの値を一方に，大きいほうの値をもう片方に出力する．図 3 は $n = 5$ のときのソーティングネットワークの 1 例であり，バブルソートにあたるものである．また，全ての比較器が上下隣接した横線にしか接続していないとき，プリミティブ・ソーティング・ネットワーク (primitive sorting network) と呼ぶ．すなわち，図 3 はプリミティブであるが，図 1 はそうではない．

このプリミティブ・ソーティング・ネットワークに関連した面白い問題として，n 個の数を整列する相異なる比較器最小のプリミティブ・ソーティング・ネットワークの総数を数え上げる問題（以降，PSN-count と呼ぶ）がある．PSN-count はランク 3 の有向マトロイドの基，順列反転の最小横棒あみだくじの数え上げ問題と等価であることが知られている．PSN-count は幾

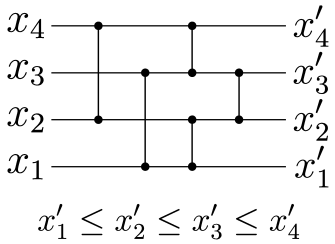

図 1　ソーティング・ネットワークの例

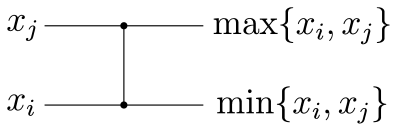

図 2　比較器

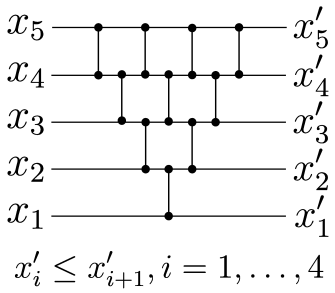

図 3　プリミティブ・ソーティング・ネットワークの例（バブルソート）

たなか　ゆうま
成蹊大学
〒 180–8633 東京都武蔵野市吉祥寺北町 3–3–1

何や代数の分野に関連が深く，Knuth の本 [2] にも紹介された興味深い問題で，2011 年には $\#\mathcal{P}$ 完全性が示されている [3]．既存研究では，PSN-count は $n = 13$ まで計算されていた [4〜6] が，著者らは $n = 14, 15$ のときの総数を新たに算出した．

2. あみだくじ数え上げ問題

あみだくじは図 4 のように n 本の縦線と，隣接する縦線間を接続するいくつかの横棒で構成させる．横棒は接続された 2 本の縦線の値を必ず反転させる（ソーティング・ネットワークの比較器とは異なることに注意）．例えば，図 4 のあみだくじの入力（上側）に順列 $(2, 3, 4, 1)$ が与えられた場合，出力（下側）は $(1, 4, 3, 2)$ になる．ここから，左から i 番目と $i+1$ 番目の縦線間の横棒を $[i : i+1]$ と表すことにする．任意のあみだくじは横棒列 $\sigma = [i_1 : i_1 + 1][i_2 : i_2 + 1]\cdots$ で表現することが可能である．横棒列 σ の左の横棒から入力された順列に適用されることを表している．ただし，この横棒列によるあみくじの表現方法を 1 通りではないことに注意しなければならない．例えば，図 4 のあみだくじの場合，[1 : 2] [3 : 4] [2 : 3] [1 : 2] [3 : 4] [2 : 3] や，[3 : 4] [1 : 2] [2 : 3] [3 : 4] [1 : 2] [2 : 3] 等の表現が可能である．もし，ある横棒列 σ の隣接した横棒 $[i_k : i_k + 1][i_{k+1} : i_{k+1} + 1]$ の順序を交換したとしても出力側で得られる順列が変わらない場合，元の横棒列 σ とその交換で得られた横棒列 σ' は同型のあみだくじと呼ぶ．逆に任意の 2 つの横棒列 σ と σ' が与えられたとき，出力の順列が変わらないように隣接した横棒同士を交換することによって σ から σ'（またはその逆）に変換できないならば，横棒列 σ と σ' は異なるあみだくじと呼ぶ．直感的には，2 つの横棒列 σ と σ' が表すあみだくじの横棒を上側にできるだけ詰めて図示したときに，それらが同じ形状をしていれば同型のあみだくじ，異なる形状をしていれば異なるあみだくじとなる．

1 つのあみだくじに対して複数の表現方法があるのは不便である．そこで，ある横棒列 σ に対して，交換しても出力側で得られる順列が変わらない全ての隣接した横棒 $[i_k : i_k + 1][i_{k+1} : i_{k+1} + 1]$ が $i_k < i_{k+1}$ を満たす場合，その横棒列 σ を representative form と呼ぶことにする．例えば，図 4 のあみだくじの representative form は [1 : 2][3 : 4][2 : 3][1 : 2][3 : 4][2 : 3] である．

本稿で取り扱うあみだくじ数え上げ問題とは，順列 $(n, n-1, \ldots, 1)$ を $(1, 2, \ldots, n)$ に置換する，相異なる横棒数最小の順列反転あみだくじの総数を求める問題である．この問題は前述したとおり PSN-count と等価な問題である．2 つの問題の等価性については Floyd の証明[1]から簡単に導き出せる．

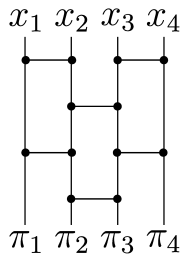

図 4 あみだくじの例（出力ベクトルが $\pi = (\pi_1, \pi_2, \pi_3, \pi_4)$）

3. 準備

アルゴリズムを説明する前にいくつかの準備を行う．n 本の縦線を持つあみだくじの入力側には常に順列 $(n, n-1, \ldots, 1)$ が与えられるとする．任意のあみだくじの出力側で得られた順列をベクトル $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ で表す．便宜上，任意の出力ベクトル $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ に対して，$\pi(i) = \pi_i$ とする．任意のあみだくじの 1 番下から横棒 $[i : i+1]$ を新たに追加することは，出力ベクトルの隣り合った 2 つの要素 $\pi(i)$ と $\pi(i+1)$ を入れ替えることに対応する．

ここで，出力ベクトル π に対する逆転数 $inv(\pi)$ を，全ての $i, j \in \{1, 2, \ldots, n\}$ に対する $i < j$ かつ $\pi(i) > \pi(j)$ を満たすペア (i, j) の総数と定義する．例えば，$n = 4$ かつ $\pi = (4, 2, 3, 1)$ のとき，$inv(\pi) = 5$ となる．逆転数の定義より，任意のあみだくじの 1 番下から横棒を追加すると $inv(\pi)$ がちょうど 1 だけ変化する．横棒が全くない（縦線だけの）あみだくじの出力ベクトル $(n, n-1, \ldots, 1)$ の逆転数が $n(n-1)/2$ であり，出力ベクトル $(1, 2, \ldots, n)$ の逆転数が 0 であることに注意すると，順列 $(n, n-1, \ldots, 1)$ を $(1, 2, \ldots, n)$ に置換するあみだくじは，少なくとも $n(n-1)/2$ の横棒が必要であることがわかる．また，$n(n-1)/2$ 本の横棒を持つそのようなあみだくじの 1 つを構成することはたやすくできる（例として図 3 のようなバブルソートに対応するあみだくじがある）ので，順列 $(n, n-1, \ldots, 1)$ を $(1, 2, \ldots, n)$ に置換するあみだくじの最小横棒数は $n(n-1)/2$ である．

[1] Knuth の本 [2] に Floyed が口頭で証明したと記述がある．証明自体は [7] に記載されている．

4. アルゴリズム

著者らが提案したアルゴリズムはフロンティア法に基づくものである．フロンティア法は Knuth が提唱した SIMPATH アルゴリズム [8] を岩下らが一般化した算法 [9] であり，ZDD (Zero-suppress Binary Decision Diagram) を高速に構築する方法として知られている[2]．

著者らのアルゴリズムの基本的なアイデアは非常に単純なものである．横棒のないあみだくじから始め，あみだくじの 1 番下側の各位置に横棒を次々に追加するという操作を，出力ベクトルが $(1, 2, \ldots, n)$ になるまで繰り返すというものである．この操作は図 5 のような木構造を構築することに対応する．木の各頂点は構築途中のあみだくじを，根は横棒のないあみだくじ，各葉は出力ベクトルが $(1, 2, \ldots, n)$ であるあみだくじを表す．各頂点には，構築途中のあみだくじの出力ベクトルを持たせる．各辺は親頂点のあみだくじの 1 番下側の各位置に横棒を追加することで生成される子頂点のあみだくじを表している．図 5 の例では，根（横棒のないあみだくじ）から 3 通りの構築途中のあみだくじが生成できることを示している．

横棒を追加して新たな頂点を作成するとき，最終的に得られるあみだくじの横棒数が最小であり，同型のあみだくじが複数出現しないようにしなければならない．例えば，図 5 の上から 2 段目 1 番左の出力ベクトル (3, 4, 2, 1) の頂点に対して $[1:2]$ の横棒を追加したとすると，明らかに最終的に横棒数が最小になり得ない（出力ベクトルが根と同じ (4, 3, 2, 1) になってしまう）．また，3 段目左から 3 番目の頂点と，6 番目の頂点は同型のあみだくじになってしまっているため，どちらか一方を生成してはいけない．

そこで，出力ベクトル π を持つ頂点から新たに横棒 $[i:i+1]$ を追加して新たな子頂点を生成するとき，下記のようなルールを設ける．

横棒挿入ルール

(1) $\pi(i) > \pi(i+1)$ を満たす．

(2) 直前に追加された（親頂点で追加された）横棒が $[j:j+1]$ であるとき，$i \geq j-1$ を満たす

ルール (1) は最終的に得られるあみだくじの横棒数が最小であることを保証するためのものであり，これは横棒を追加したときに常に逆転数 $inv(\pi)$ が 1 減らなければ，最小本数のあみだくじを構成できないという性質からきている．この性質は $(n, n-1, \ldots, 1)$ の逆転数が $n(n-1)/2$ かつ最小横棒数が $n(n-1)/2$ であることから明らかである．ルール (2) は同型のあみだくじを生成しないためのものであり，これは representative form を満たす横棒の追加しか行えなくするためのものである．直前に追加された（親頂点で追加された）横棒が $[j:j+1]$ であるとき，$i < j-1$ を満たす横棒 $[i:i+1]$ を新たに追加してしまうと，$[j:j+1]$ と $[i:i+1]$ は交換しても同型のあみだくじであり，representative form を満たす横棒の追加とならない．一方，その他の場合は異なるあみだくじか，representative form を満たす横棒の追加となる．

著者らのアルゴリズムの基本的枠組みを以下に示す．

Algorithm_PSN_count1

入力：正整数 n

出力：縦線 n 本の順列反転かつ横棒最小あみだくじの総数

---

1. A_0 を横棒のないあみだくじを表す出力ベクトル $\pi = (n, n-1, \ldots, 1)$ とダミーの横棒 $[0, 1]$ のペア $\langle \pi, [0, 1] \rangle$ だけを元として持つ多重集合とし，$k := 0$ とする．

2. A_{k+1} を空の多重集合とする．

3. A_k の出力ベクトル π と直前に追加した横棒 $[j, j+1]$ の全てのペアに対して，各縦線の間の下側からルール 1 とルール 2 を満たす横棒を 1 本追加し，新たに作成された m 個のあみだくじを表す出力ベクトル $\pi^1, \pi^2, \ldots, \pi^m (m < n)$ を生成する．生成された全ての出力ベクトル $\pi^\ell (\ell = 1, \ldots, m)$ と，そのとき追加した横棒 $[i^\ell, i^\ell + 1]$ に対して，ペア $\langle \pi^\ell, [i^\ell, i^\ell + 1] \rangle$ を A_{k+1} に挿入する．

4. $k := k + 1$ とする．

5. $k = n(n-1)/2$ ならば A_k の要素数を出力して終了する．そうでなければ，2 に戻る．

Algorithm_PSN_count1 は正しくあみだくじの総数を数え上げる．しかし，$A_{n(n-1)/2}$ の要素数はあみだくじの総数と同じとなるため，これをそのまま実装し実行するのは現実的でない．そこで，同じ状態であると見なしてもよい頂点をまとめることを考える．これはフロンティア法のキモになる部分である．図 5 の頂点 A と頂点 B は同じ出力ベクトルを持ち，追加可能な横棒の位置が（ルール (1) と (2) より）同じであることがわかる．このとき，図 6 のように頂点 A と頂点 B

[2] ZDD およびフロンティア法については OR 学会機関誌 2012 年 11 月号に詳しく解説されている [10] ので，興味のある読者は参照されたい．

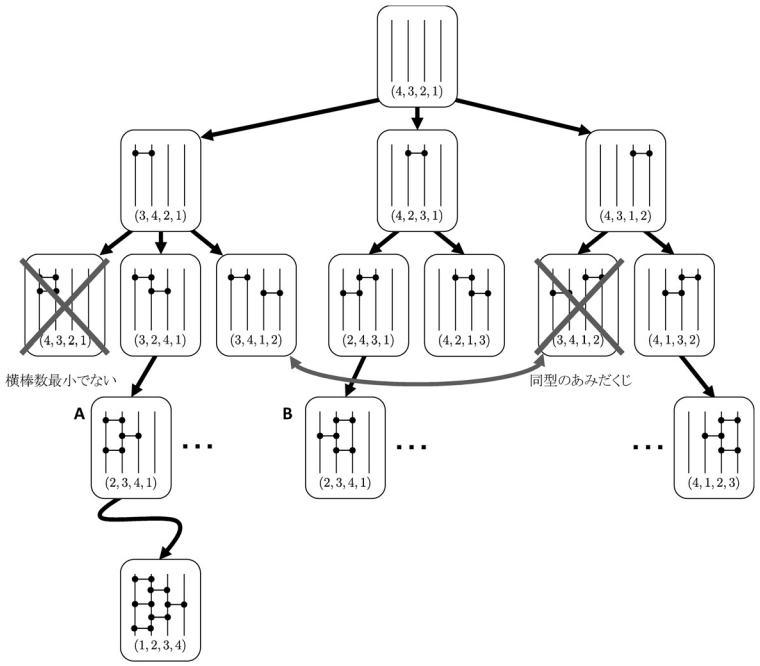

図 5　あみだくじの 1 番下側の各位置に横棒を追加することで構築される木構造

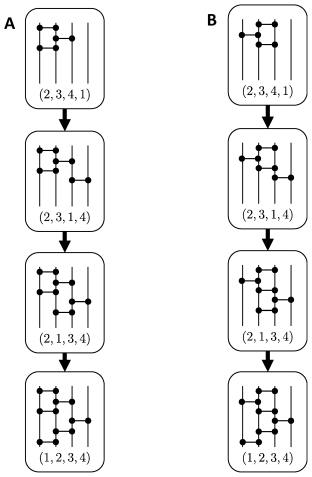

図 6　頂点 A と頂点 B の部分木

を根とする部分木を考えると，その構造が同じ（その後に追加可能な横棒の位置が同じ）であることに気づく．すなわち，出力ベクトルが同じかつ追加可能な横棒の位置が同じである頂点は（数え上げるという意味では）まとめてしまって何ら問題がない．頂点をまとめる場合はいくつの頂点がまとまっているかがわかるように，各頂点に頂点がまとまっている数を持たせる．

Algorithm_PSN_count1 を改良したアルゴリズムを下記に示す．

Algorithm_PSN_count2

入力：正整数 n

出力：縦線 n 本の順列反転かつ横棒最小あみだくじの総数

1. A_0 を横棒のないあみだくじを表す出力ベクトル $\pi = (n, n-1, \ldots, 1)$, ダミーの横棒 $[0, 1]$ とまとめた頂点数のタプル $\langle \pi, [0, 1], 1 \rangle$ だけを元として持つ多重集合とし，$k := 0$ とする．
2. A_{k+1} を空の多重集合とする．
3. A_k の全ての出力ベクトル π, 直前に追加した横棒 $[j, j+1]$ とまとめた頂点数 $count$ のタプルに対して，各縦線の間の下側からルール 1 とルール 2 を満たす横棒を 1 本追加し，新たに作成された m 個のあみだくじを表す出力ベクトル $\pi^1, \pi^2, \ldots, \pi^m (m < n)$ を生成する．生成された全ての出力ベクトル $\pi^\ell (\ell = 1, \ldots, m)$ と，そのとき追加した横棒 $[i^\ell, i^\ell + 1]$ に対して，タプル $\langle \pi^\ell, [i^\ell, i^\ell + 1], count \rangle$ を A_{k+1} に挿入する．
4. A_{k+1} の要素の中で出力ベクトルが同じかつ次に横棒が追加可能な位置が同じである頂点をまとめる．例えば，タプル $\langle \pi^\alpha, [i^\alpha, i^\alpha + 1], count^\alpha \rangle$ と $\langle \pi^\beta, [i^\beta, i^\beta + 1], count^\beta \rangle$ がまとめられる場合，2 つのタプルを $\langle \pi^\alpha, [i^\alpha, i^\alpha + 1], count^\alpha + count^\beta \rangle$

としてまとめる．

5. $k := k+1$ とする．

6. $k = n(n-1)/2$ ならば A_k の唯一の要素であるタプルのまとめた頂点数 *count* を出力して終了する．そうでなければ，2 に戻る．

プログラミングが得意な人であれば Algorithm_PSN_count2 を教えれば，おそらく実装することは難しくないだろう．また，フロンティア法の枠組みを心得ている人であればAlgorithm_PSN_count2 を思いつくこともそれほど難しくはないだろう．しかし，実装する人によっておそらく相当な計算時間の差が生まれるだろう．なぜならば，Algorithm_PSN_count2 にはまだ明確に書かれていない部分があるからである．それは **4.** の「どのようにまとめる頂点を探し出すか？」というところである．何も考えてない場合は A_{k+1} 内の全ての頂点対について比較するかもしれない．ちょっと賢い人は，2 分探索木やトライ木を使うかもしれない．フロンティア法の枠組みを心得ている者であれば，おそらくハッシュを使うだろう．しかし，ハッシュを使ったら良いということを知っていても，どのようにハッシュを実装したら良いかということまではわからない．

さらに細かいことを言うと，「頂点をどのようなデータ構造で保持するのが良いのか？」，「メモリ管理はどのようにしたら良いのか？」ということもわからない．アルゴリズムを実装する方法には自由度がかなりあり，作成者が自由に選択できる箇所があることがそれなりにある．そのため，筆者は擬似コード等で書かれたアルゴリズムを見ただけで満足するのではなく，実際にアルゴリズムを組み上げることによって本当の理解が得られるのではと考えている．今回の成果も一度のプログラミングでうまくいったわけではなく，繰り返しプログラミングすることによって高速に動作できるようになった．プログラミングによって得られる知見というのもそれなりにあると筆者個人は考えている．

さて，筆者自身はどのようにこの問題を解決したのかだが，まず出力ベクトル π をキーにした適当なハッシュ関数をつくり，ハッシュ値が同じ頂点をリストで管理するようにした．実は n 要素の全ての順列 σ ($(1,2,\ldots,n)$ から $(n,n-1,\ldots,1)$ まで）は次の式によって通し番号を付けることが可能である．

$$\begin{aligned} id(\sigma) = & small_\sigma(1)\cdot(n-1)! + small_\sigma(2)\cdot(n-2)! \\ & + \cdots + small_\sigma(n) * 0! \end{aligned}$$

ここで，$small_\sigma(i) = \left|\{j \mid i < j < n \text{ かつ順列}\sigma\text{の}i\text{番目の要素より}j\text{番目の要素のほうが小さい}\}\right|$ である．この $id(\sigma)$ と適当な整数値 M を用いて，出力ベクトル π に対して次のようにハッシュ関数を定める．

$$hash(\pi) = id(\pi) \mod M$$

このハッシュ関数を用いて実装してみると，うまく動作することがわかった．ただし，この方法は M をどのような値にすれば適切かがすぐにはわからないという点と，頂点内容を全てリストで保持する必要があるためメモリ効率が良いとは言えない点が問題である．数え上げ問題では，計算時間がすぐに爆発してしまうことが非常に問題となるが，速度を犠牲にせずにメモリ使用量をいかに抑えるかという工夫も非常に大切である．実際，筆者もメモリ使用量を抑えることばかり考えていた．

では最終的にどうしたかというと，A_{k+1} 内にある頂点の出力ベクトル π は全て同じ逆転数 $inv(\pi)$ を持つということから，逆転数が同じである n 要素の全ての順列 σ に通し番号が付けることができないかどうかを考えた．実は逆転数が c である n 要素の全ての順列の総数 $T(n,c)$ は Mahonian の三角数と呼ばれる数列を成すことが知られており，以下の漸化式で計算することができる．

$$\begin{aligned} & T(1,0) = 1, \\ & T(1,c) = 0 \quad \text{for} \quad c \neq 0, \\ & T(n,c) = \sum_{t=0}^{n-1} T(n-1, c-t) \quad \text{for} \quad n > 1. \end{aligned}$$

詳細は省略するが，逆転数が同じである n 要素の全ての順列の総数を数え上げることができるため，A_{k+1} 内にある頂点の出力ベクトル π に対して通し番号を付けることが可能（最小完全ハッシュ関数を定義することが可能）である．そのため，この Mahonian の三角数を用いたハッシュ関数は，先ほど説明した $hash(\pi)$ と違い，（ハッシュ値を見れば出力ベクトル π を復元できるため）出力ベクトル π をメモリ上に保存する必要がなくなる．また，同じハッシュ値を保持するリストの長さが高々$n-1$ になるので（横棒を挿入できる位置が高々$n-1$ 箇所であるので），アルゴリズム全体の計算量を $\mathcal{O}(n^2 n!)$ とはっきりと述べることが可能となる．

5. 計算実験

最後に，前節で説明した **Algorithm_PSN_count2** を実際にプログラミングした計算実験結果を

表 1　計算結果

n	あみだくじの総数	河原ら [5](sec.)	提案手法 (sec.)
1	1	—	—
2	1	—	—
3	2	0.0	—
4	8	0.0	0.0
5	62	0.0	0.0
6	908	0.0	0.0
7	24698	0.1	0.0
8	1232944	0.3	0.0
9	112018190	4.1	0.0
10	18410581880	62.0	0.0
11	5449192389984	964.1	0.5
12	2894710651370536	20172.6	8.7
13	2752596959306389652	473314.0	139.2
14	4675651520558571537540	—	2605.5
15	1416380899558002221878639O	—	97334.8

紹介したい．ハッシュ関数に Mahonian の三角数を利用した計算結果を表 1 に示す．ただし，表 1 の結果には紙面の都合により説明できなかった高速化手法も組み込んである．表 1 の左の列から，縦線の数 n，あみだくじの総数，$n = 13$ までの計算を行った河原らのアルゴリズム [5] の計算時間，提案アルゴリズムの計算時間を表す．計算時間の “—”はアルゴリズムの性質上の理由で計算できないか，計算が行われていないことを表す．河原らの実験環境と著者らの実験環境は同一ではないため公平な比較ではないが，$n = 13$ のときを見ると著者らのアルゴリズムは河原らのアルゴリズムに比べて 3,000 倍の高速化に成功していることがわかる．

6. おわりに

本稿では，著者らがあみだくじ数え上げ問題に対する世界記録を更新した経緯について紹介した．このような文章を書くのが初めてということもあり，うまくお伝えできたかどうかは甚だ心細いが，アルゴリズムをどのように考え，実装中にどのようなことを考えたかを解説した．読者の皆様に実装を行うことで新たな発想が生まれる可能性があるということを理解いただければ幸いである．つい最近 $n = 16$ を計算できそうなアイデアを思いついたのだが，この内容を論文としてまとめていないので，早急に書き上げることが課題である．

参考文献

[1] Y. Tanaka, A. Ikegami, Y. Matsui, K. Fujisawa and Y. Yasui, “A fast algorithm for counting the number of primitive sorting networks,” *Proc. of the 20th Conference of the International Federation of Operational Research Societies (IFORS2014)*, 80, 2014.

[2] D. E. Knuth, *Axioms and Hulls, Lecture Notes in Computer Science*, Springer-Verlag, 1992.

[3] M. J. Samuel, “Word posets, with applications to Coxeter groups,” *Proc. of the 8th Internatinal Conference on WORDS*, 226–230, 2011.

[4] K. Yamanaka, S. Nakano, Y. Matsui, R. Uehara and K. Nakada, “Efficient enumeration of all ladder Lotteries and its application,” *Theoretical Computer Science*, **411**, 1714–1722, 2010.

[5] J. Kawahara, T. Saitoh, R. Yoshinaka and S. Minato, “Counting primitive sorting networks by PiDDs,” Division of Computer Science Report Series A, Graduate School of Information Science and Technology, Hokkaido University, Technical Report, TCS-TR-A-11-54, 2011.

[6] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences, http://oeis.org/A006245 (accessed October 2, 2014)

[7] D. E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*, 2nd ed., Addison-Wesley Professional, 1998.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*, 1st ed., Addison-Wesley Professional, 2011.

[9] H. Iwashita, J. Kawahara and S. Minato, “ZDD-based computation of the number of paths in a graph,” Division of Computer Science Report Series A, Graduate School of Information Science and Technology, Hokkaido University, Technical Report, TCS-TR-A-12-60, 2011.

[10] 湊真一，“BDD ／ ZDD を用いた新しい列挙牽引化技法（フロンティア法）とその応用，”オペレーションズ・リサーチ，**57**, 596–628, 2012.

