ELSEVIER

Theoretical Computer Science 159 (1996) 15–28

Theoretical Computer Science

A parallel algorithm for the generation of a permutation and applications

Laurent Alonso *, René Schott [1]

CRIN, INRIA-Lorraine, Université de Nancy 1, 54506 Vandoeuvre-lès-Nancy, France

Abstract

A parallel algorithm is presented for generating a permutation of size n. The algorithm uses $O(n)$ processors and runs in $O(\text{Log}^2(n))$ time. We show also that this algorithm permits the generation of a Dyck word. The same techniques work for Motzkin words, left factors of Dyck or Motzkin words and words which are in bijection with trees split into patterns as defined by Dershowitz and Zaks (1989) (see Alonso, 1992).

1. Introduction

Due to numerous applications in graphics, statistics and engineering, algorithms for the generation of trees have been extensively studied. In recent years, a number of parallel algorithms have appeared for the generation of combinatorial objects such as permutations [8], combinations [9], subsets, equivalence relations, etc. These works concern the generation of all objects under consideration. The purpose of this paper is different since we design a parallel algorithm which generates uniformly one of these objects. This problem appears frequently in computer graphics and in statistics (for sequential algorithms, see e.g. [3, 6, 12, 15]).

We present in this paper a method to generate uniformely a random permutation of size n in time $O(\text{Log}^2(n))$ using $O(n)$ processors (another method for generating permutation on a shared memory machine can be found in [5]). This algorithm has some useful extensions, indeed we can use it for generating randomly a Dyck word of size n or a left factor of a Dyck word. Moreover, it can also be used to generate more complicated objects such as Motzkin words or left factors of Motzkin words. Even if the great challenge is to find efficient sequential algorithms for the generation of these two structures [4, 6], we show here that these two objects can be efficiently generated

* Corresponding author. Email: alonso@loria.fr.
[1] Email: schott@loria.fr.

SSDI 0304-3975(95)00198-0

on a parallel machine and we present two algorithms which generate them in average time $O(\mathrm{Log}^2(n))$ with $O(n)$ processors.

The organization of the paper is as follows: Section 2 describes the generation of a permutation. The generation of a Dyck word is detailed in Section 3 while Section 4 provides extensions to a Motzkin word, left factor of Dyck and Motzkin words. Other applications (generation of sequences in bijection with trees split into patterns) of this method can be found in [3]. Our conclusions and further aspects are offered in Section 5.

2. Generation of a permutation

Definition 1. We call lower-exceeding sequence a sequence of integers $(s_1, s_2, \ldots, s_n)$ such that

$$\forall i \quad 1 \leqslant s_i \leqslant i.$$

We remember first how the classical bijection between permutations and lower-exceeding sequences works, then we show how to implement it on a parallel machine with an average time complexity in $O(\mathrm{Log}^2(n))$. More details about this bijection are available in [20].

2.1. Bijection $\mathscr{H}$ between permutations and lower-exceeding sequences

The mapping $\mathscr{H}$ works as follows:

- It transforms a permutation of $\mathscr{S}_n$ into an $n \times n$ matrix [10] such that each row and column contains a circle. We put a circle in the box (i,j) if and only if the permutation transforms i into j.

Example. The permutation $(1,3,5,4,2)$ gives the matrix in Fig. 1.

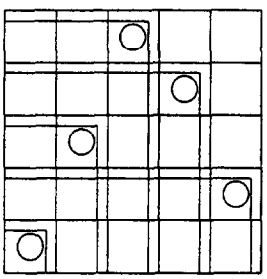


Fig. 1

- the mapping transforms then this array into a lower-exceeding sequence. This is easily done by searching the circle in position (i,j) and by associating the number

of circles located in the southwest quarter (with respect to the box (i,j)) to the ith term of our sequence.

Example. The preceding array is transformed into the array shown in Fig. 2.

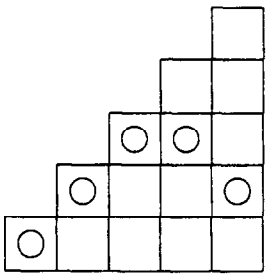


Fig. 2

This gives the lower-exceeding sequence $(1,2,3,3,2)$.

Now we define in a more formal way this application $\mathscr{H}$.

Definition 2. Let $w=(w_1,w_2,\ldots,w_j)$ be a permutation, then $s=(s_1,s_2,\ldots,s_j)=\mathscr{H}(w)$ if and only if

$$\forall i,\quad s_i = card(\{1 \leqslant k \leqslant i \text{ such that } w_k \leqslant w_i\}).$$

In the next subsection we prove that $\mathscr{H}$ is a bijection by constructing its reciprocal mapping.

2.2. *Definition of* $\mathscr{F}=\mathscr{H}^{-1}$

We have the following theorem which allows to compute $\mathscr{F}=\mathscr{H}^{-1}$:

Theorem 1. *Let* $w=(w_1,w_2,\ldots,w_j)$ *be a permutation,* i *an integer of* $[\![1,j]\!]$ *and* $s_l = card(\{1 \leqslant k \leqslant i,\ w_k \leqslant w_i\})$ *then* w_i *is the* s_l*th smallest element of* $\mathbf{N}^* - \{w_{i+1},w_{i+2},\ldots,w_j\}$.

Proof. Since w is a permutation, we can write

$$\begin{aligned} w_i &= card(\{1 \leqslant k \leqslant j,\ w_k \leqslant w_i\}) \\ &= card(\{1 \leqslant k \leqslant i,\ w_k \leqslant w_i\}) + card(\{i < k \leqslant j,\ w_k \leqslant w_i\}) \\ &= s_l + card(\{i < k \leqslant j,\ w_k < w_i\}). \end{aligned}$$

Now let q be the $s_l th$ smallest element of $\mathbf{N}^* - \{w_{i+1}, w_{i+2}, \ldots, w_j\}$, we have

$$q = card(\{v \in \mathbf{N}^*,\ v \leqslant q\})$$

$$= card(\{v \in \mathbf{N}^* - \{w_{i+1}, \ldots, w_j\},\ v \leqslant q\}) + card(\{v \in \{w_{i+1}, \ldots, w_j\},\ v \leqslant q\})$$

$$= s_l + card(\{i < k \leqslant j,\ w_k < w_i\}) = w_i. \qquad \square$$

This theorem is very important. Indeed it explains how to build a permutation $(w_1, w_2, \ldots, w_j)$ from the sequence $(s_1, s_2, \ldots, s_j) = \mathscr{H}((w_1, w_2, \ldots, w_j))$. For that purpose, we need only to take an array $j \times j$ and to add a circle in each column from the right to left by adding a circle in the s_ith unused row of each column i.

For instance, we obtain with $(s_1, \ldots, s_5) = (1, 2, 1, 4, 3)$ the arrays shown in Fig. 3.

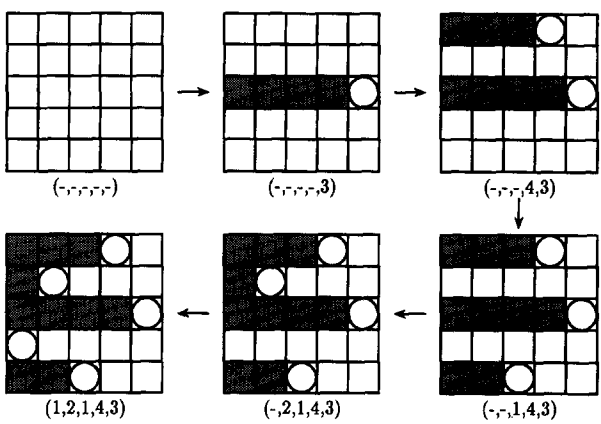


Fig. 3

It is worth noting that when we cut the final array in two parts, we get the arrays shown in Fig. 4.

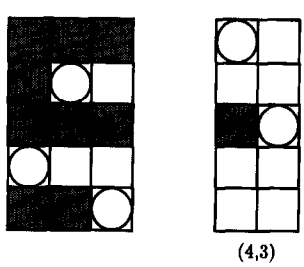


Fig. 4

If we remove the completely gray row in the left array (see Fig. 5),

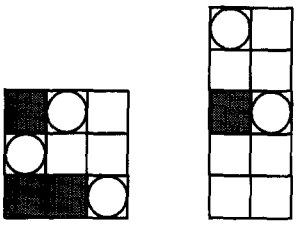


Fig. 5

we obtain two arrays which correspond, respectively, to the sequences $(s_1,\ldots,s_i)$ and $(s_{i+1},\ldots,s_j)$.

We use this recursive decomposition of the problem to compute efficiently the function $\mathscr{F} = \mathscr{H}^{-1}$. Indeed, in order to calculate $\mathscr{F}((s_i,\ldots,s_j))$, we compute first $\mathscr{F}((s_i,\ldots,s_{\lfloor(i+j)/2\rfloor}))$ and $\mathscr{F}((s_{\lfloor(i+j)/2\rfloor+1},\ldots,s_j))$ (or in an equivalent way, we build their two corresponding arrays). Then we modify the first resulting array.

In fact, this modification requires only to know the unordered set of the integers which appear in $\mathscr{F}((s_{\lfloor(i+j)/2\rfloor+1},\ldots,s_j))$ (or in an equivalent way the set of the used rows of the array corresponding to $(s_{\lfloor(i+j)/2\rfloor+1},\ldots,s_j)$) and to avoid to use these numbers.

For instance, if we recall again the last obtained array (see Fig. 5), we see that the third and fifth rows are used in the right array. Thus in order to build the first part of the array which is of interest, we take a new array in which the third and fifth rows are gray rows. Then we add, respectively, a circle in the second, third, first unused white rows of the first, second, third column of this array. We get the matrix shown in Fig. 6.

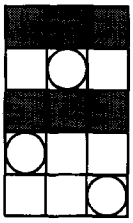
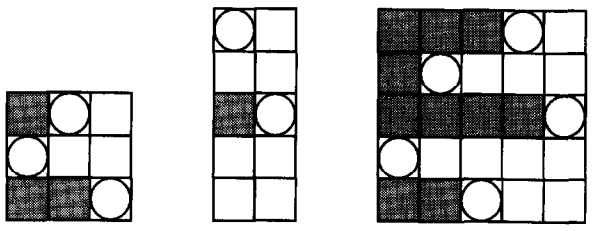


Fig. 6

To compute efficiently the rows in which we add a circle, it is worth sorting the elements of $\mathscr{F}((s_{\lfloor(i+j)/2\rfloor+1},\ldots,s_j))$. This explains why we compute a new function $\mathscr{G}(i,(s_i,\ldots,s_j)) = ((x_i,a_i),\ldots,(x_j,a_j))$. This function gives the sorted list $(x_i,\ldots,x_j)$

of $(w_i,\ldots,w_j) = \mathscr{F}((s_i,\ldots,s_j))$ and the numbers of the columns $a_i,\ldots,a_j$ in which a circle appears, respectively in the x_ith, ..., x_jth rows.

For the previous example we have the matrices shown in Fig. 7.



$\mathscr{G}(1,(1,2,1)) = ((1,3),(2,1),(3,2))$ $\mathscr{G}(4,(4,3)) = ((3,5),(5,4))$ $\mathscr{G}(1,(1,2,1,4,3)) = ((1,3),(2,1),(3,5),(4,2),(5,4))$

Fig. 7

Assume that we know:

- $\mathscr{G}(i,(s_i,\ldots,s_j)) = ((x_i,a_i),\ldots,(x_j,a_j))$,
- $\mathscr{G}(j+1,(s_{j+1},\ldots,s_l)) = ((x_{j+1},a_{j+1}),\ldots,(x_l,a_l))$,

and that we want to compute $\mathscr{G}(i,(s_i,\ldots,s_l)) = ((y_i,b_i),\ldots,(y_l,b_l))$. We can deduce the following lemma from the preceding discussion:

Lemma 1.

- *For each k such that $j < k \leqslant l$, there exists an integer v such that $(x_k,a_k) = (y_v,b_v)$,*
- *For each k such that $i \leqslant k \leqslant j$, denote by z_k the x_kth element of $\mathbf{N}^* - \{x_{j+1}, x_{j+2},\ldots,x_l\}$. Then there exists an integer v such that $(y_v,b_v) = (z_k,a_k)$.*

We can deduce from this lemma two theorems which allow us to build $\mathscr{G}(i,(s_i,\ldots,s_l))$ from $\mathscr{G}(i,(s_i,\ldots,s_j))$ and $\mathscr{G}(j+1,(s_{j+1},\ldots,s_l))$ by merging two sorted lists.

Theorem 2. *Let m and n be two integers between i and l and consider the integers o and p such that $a_m = b_o$ and $a_n = b_p$. Then $o < p$ if and only if:*

- *if $m \leqslant j$ and $n \leqslant j$, then $m < n$,*
- *if $m > j$ and $n > j$, then $m < n$,*
- *if $m \leqslant j$ and $n > j$, then $x_m + (n-j) \leqslant x_n$,*
- *if $m > j$ and $n \leqslant j$, then $x_n + (m-j) \leqslant x_m$.*

Proof. The first two relations follow directly from Lemma 1. The last two cases are equivalent if we replace n by m, we can therefore restrict our study to the case where $m \leqslant j$ and $n > j$. Assume that $m \leqslant j$ and $n > j$, then:

- y_p is equal to x_n,
- y_o corresponds to the x_m*th* element of $\mathbf{N}^* - \{x_{j+1},\ldots,x_l\}$.

The inequality $y_p > y_o$ is verified if and only if there are more than x_m elements in $[1, x_n[- \{x_{j+1}, x_{j+2}, \ldots, x_{n-1}\}$ (i.e. if and only if $x_m \leqslant x_n - 1 - (n - 1 - j)$). □

This theorem gives means for computing the new position of the image of a pair (x_p, a_p) in $\mathscr{G}(1, (s_1, \ldots, s_l))$ by doing only comparisons between the pairs (x_m, a_m) and (x_n, a_n). Assume, in addition, that we succeed in reordering the pairs (x_m, a_m) with the order induced by Theorem 2, we can then go quickly from this representation to $((y_i, b_i), \ldots, (y_l, b_l))$. In fact, it is sufficient to transform all pairs (x_m, a_m) according to the following rule:

Theorem 3. *Let n be the new position of the pair (x_m, a_m),*
- *if $a_m > j$ then $(y_n, b_n) = (x_m, a_m)$,*
- *if $a_m \leqslant j$ then $(y_n, b_n) = (x_m + n - m, a_m)$.*

Proof. If $a_m > j$, we use Lemma 1 in order to conclude. Assume now that $a_m \leqslant j$, we have therefore:

$$
\begin{aligned}
n - i + 1 &= card(\{i \leqslant k \leqslant n\}) \\
&= card(\{i \leqslant k \leqslant n,\ y_k \leqslant y_n\}) \\
&= card(\{i \leqslant k \leqslant n,\ b_k \leqslant j \text{ and } y_k \leqslant y_n\}) \\
&\quad + card(\{i \leqslant k \leqslant n,\ b_k > j \text{ and } y_k \leqslant y_n\}) \\
&= card(\{i \leqslant k \leqslant j,\ x_k \leqslant x_m\}) \\
&\quad + card(\{k > j \text{ such that } x_k \text{ is less than the } x_m\text{th element of } \mathbf{N}^* \\
&\quad - \{x_{j+1}, \ldots, x_l\}\}) \\
&= n - i + 1 \\
&\quad + card(\{k > j \text{ such that } x_k \text{ is less than the } x_m\text{th element of } \mathbf{N}^* \\
&\quad - \{x_{j+1}, \ldots, x_l\}\})
\end{aligned}
$$

but using Lemma 1, we get:

$$
\begin{aligned}
y_n &= \text{the } x_m\text{th element of } \mathbf{N} - \{x_j, \ldots, x_l\} \\
&= x_m + card(\{k > j \text{ such that } x_k \text{ is less than the } x_m\text{th element of } \mathbf{N}^* \\
&\quad - \{x_{j+1}, \ldots, x_l\}\}).
\end{aligned}
$$

Therefore $y_n = x_m + n - m$. □

2.2.1. Implementation of the algorithm

The generation of a permutation is done as follows: we generate first a lower-exceeding sequence $(s_1, \ldots, s_n)$ with the help of n processors. The ith processor chooses a random number in the interval $[1, i]$. Then, using a network which we will describe

later, we compute the value of $\mathscr{G}(1,(s_1,\ldots,s_n)) = ((1,a_1),\ldots,(n,a_n))$. We get a permutation $(a_1,\ldots,a_n)$.

Theorem 4. *All permutations $(a_1,\ldots,a_n)$ of size n are obtained with probability $1/n!$.*

Proof. The permutation obtained with our algorithm is in fact the inverse of the permutation $\mathscr{F}((s_1,\ldots,s_n))$. The probability of generating a permutation w is therefore equal to the probability of generating, at the beginning, a lower-exceeding sequence $\mathscr{F}^{-1}(w^{-1}) = \mathscr{H}(w^{-1})$, this means $1/n!$. □

It remains to find a processor network which is able to compute the function $\mathscr{G}$. In fact, we can take all processor networks which sort a list of n elements by implementing the merging sort. Indeed, if we have a processor which can merge two sorted lists of size k and if we know $\mathscr{G}(i,(s_i,\ldots,s_{i+k-1}))$, $\mathscr{G}(i+k,(s_{i+k},\ldots,s_{i+2k-1}))$, then this processor computes the value of $\mathscr{G}(i,(s_i,\ldots,s_{i+2k-1}))$.

We assume for simplicity that $n = 2^p$, and that we have a processor $\mathscr{N}(p)$ which can merge two sorted lists of 2^{p-1} elements. The network given in Fig. 8 does the job:

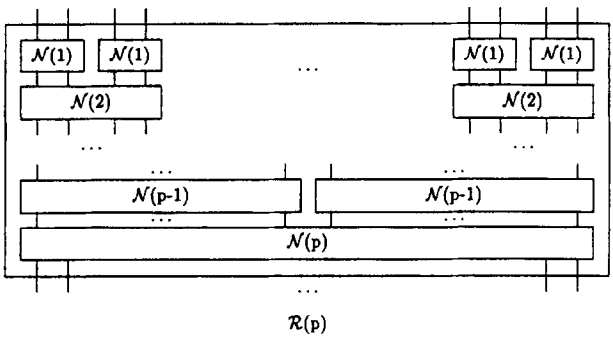


Fig. 8

Here $\mathscr{N}(P)$ is a network which is able to merge two sorted lists with the help of Theorem 2. Then it uses the results of Theorem 3 in order to get the value of $\mathscr{G}$.

There exists a network which is able to merge two sorted lists of size 2^{p-1} in time $O(p)$ with approximately 2^p processors (see [7, 18] for more details). If we use this network, we have a full network with $O(n\ \mathrm{Log}(n))$ processors but this network can be neatly implemented with $O(n)$ processors using a shuffle-exchange network ([19]). Such a network transforms a lower-exceeding sequence s into $\mathscr{F}(s)^{-1}$, and solves our problem in time $O(\mathrm{Log}^2(n))$.

3. Generation of a Dyck word

Definition 3. Let $A = \{x, y\}$ be a set alphabet. The Dyck language D is defined by the production rule $D \to \varepsilon + xDyD$ where ε is the empty word.

In order to generate a Dyck word of size $2n$, first recall how a sequential algorithm can build a random Dyck word in time $O(n)$ [3]. This algorithm works in three steps:

- generation of a random permutation of length $2n+1$,
- transformation of this permutation into a sequence of $n+1$ letters x and n letters y,
- transformation of this sequence into a Dyck word of length $2n$.

We have shown in the previous section how to get a random permutation of size $2n+1$. We replace now in this permutation the numbers strictly greater than n by the letter x and the others by the letter y in order to get a random word with $n+1$ letters x and n letters y spread out on $2n+1$ processors.

We show here how to transform it into a 1-dominating sequence (i.e. a letter x followed by a Dyck word). Let $u_0 \dots u_1 \dots u_{2n}$ be the word composed with letters x and y and define the functions:

- $f(i,j) = card(\{i \leqslant l \leqslant j,\ u_l = x\}) - card(\{i \leqslant l \leqslant j,\ u_l = y\})$,
- $g(i,j)$ the last position where the minimum of the function $v \mapsto f(i,v)$ appears in the interval $[\![i,j]\!]$,
- $h(i,j)$ the value of the minimum of the function $x \mapsto f(i,x)$ in the interval $[\![i,j]\!]$.

The value $g(0,2n)$ permits us to know where to perform the cyclic transformation which will transform the word composed with x's and y's into a 1-dominating word.

3.1. Computation of $g(0,2n)$

The function g is strongly related to f and h and their computation will be done by induction at the same time. In fact:

Proposition 1. *For f, g and h when $i < j < l$:*

- $f(i,i) = 1$ *if* $u_i = x$ *and* $f(i,i) = -1$ *if* $u_i = y$,
- $g(i,i) = i$,
- $h(i,i) = f(i,i)$,
- $f(i,l) = f(i,j) + f(j+1,l)$,
- $h(i,l) = \min(h(i,j), f(i,j) + h(j+1,l))$,
- $g(i,l) = \begin{cases} g(i,j) & \text{if } h(i,j) < f(i,j) + h(j+1,l), \\ g(j+1,l) & \text{if } h(i,j) \geqslant f(i,j) + h(j+1,l). \end{cases}$

Proof. The first three properties follow directly from the definitions of f, g, h. The proof of the other is by induction. □

This property permits to compute recursively the values of f, g, h. We will describe a network using $O(n)$ processors which does the job in time O(Log(n)). For simplicity we assume that $2n+1$ is a power of 2 (this assumption does not change the results). The network is a perfect binary tree with $2n+1$ leaves which are the processors containing the terms of the sequence $(u_0, u_1, \ldots, u_{2n})$. The values of f, g and h are computed for each leaf, then the values of f, g and h are computed with the parameters $(2k, 2k+1)$, $(4k, 4k+3)$, ..., $(0, 2n)$ one step bottom up in the tree.

Example. See Fig. 9.

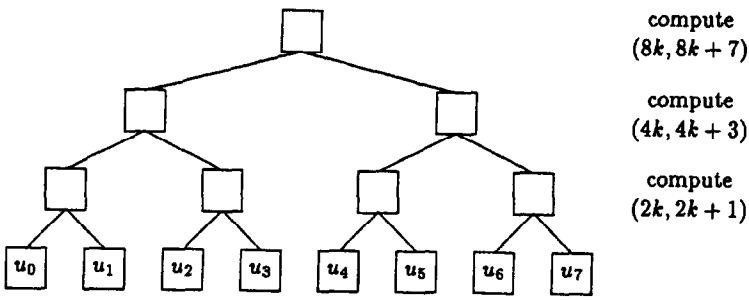


Fig. 9

As soon as the value of $g(0, 2n)$ is computed, we can propagate this result in time $O(\mathrm{Log}(n))$ so that each processor knows where the cyclic permutation has to be performed.

3.2. Cyclic permutation

We apply the cycle lemma [14], with the value $N = g(0, 2n)$ as pivot. This brings the terms $(u_{N+1}, \ldots, u_{2n})$ in position $(u_0, \ldots, u_{2n-N-1})$ and the terms $(u_0, \ldots, u_N)$ in position $(u_{2n-N}, \ldots, u_{2n})$ and can be done with a sorting network, because we know in which places the values of the elements have to be. These numbers are used as keys on which we will do the sorting. This leads to an algorithm whose time complexity is in $O(\mathrm{Log}(n))$ if we use the sorting network proposed in [1] or in $O(\mathrm{Log}^2(n))$ with a merge sorting network [18]. These are $O(n)$ processor networks.

4. Extensions

The techniques developed previously apply also to the parallel generation of a Motzkin word, left factor of Dyck or Motzkin words and sequences which are in bijection with trees split into patterns as defined in [13] (see [3]). We show below

how to design a parallel algorithm for the generation of a Motzkin word and we give some hints to generate a left factor of Dyck or Motzkin words.

4.1. Motzkin words

Definition 4. Let $A = \{a, x, y\}$ be a set alphabet. The Motzkin language M is defined by the production rule $M \to \varepsilon + aM + xMyM$ where ε is the empty word.

4.1.1. Principles of the sequential algorithm

A complete description of this algorithm is in [3] and [4]. Here we remember just its main principles.

In order to generate a random Motzkin word of size $n-1$:

(i) We generate $\lceil 2n/3 \rceil$ random bits (0 or 1) and we call k the number of 1 bits.

(ii) Then, we accept or reject this choice k, depending on a random outcome.

- If $k = 0$ or $k > \lfloor (n+1)/2 \rfloor$, then we reject this choice.
- If $1 \leqslant k \leqslant \lfloor n/3 \rfloor$, then we accept the choice of k with probability $\binom{a_k^n}{c_k^n} \Big/ \binom{b_k^n}{c_k^n}$ where $a_k^n = \lfloor n/3 \rfloor$, $b_k^n = n+1-2k$, $c_k^n = \lfloor n/3 \rfloor + 1 - k$,
- If $\lfloor (n+1)/2 \rfloor \geqslant k > \lfloor n/3 \rfloor$, then we accept the choice of k with probability $\binom{a_k^n}{c_k^n} \Big/ \binom{b_k^n}{c_k^n}$ where $a_k^n = \lceil 2n/3 \rceil - k$, $b_k^n = k-1$, $c_k^n = k - 1 - \lfloor n/3 \rfloor$.

If the choice of k is rejected, we go back to the beginning.

(iii) we need to draw a random Motzkin word with $k-1$ letters x over a total of $n-1$ letters.

- First, we draw a random permutation of size n;
- then we replace the values of the permutation which are in $[\![1, k]\!]$ by the letter x, those which are in $[\![k+1, 2k-1]\!]$ by y and the remaining values by a;
- finally, we apply on this word the cyclic permutation which transforms this word into xM where M is a Motzkin word of size $n-1$.

This algorithm can be easily parallelized as we will see below.

4.1.2. A parallel algorithm for the generation of a Motzkin word

First we choose a sequence of $\lceil 2n/3 \rceil$ adjacent bits. This needs $t = \lceil 2n/3 \rceil$ processors, each of them chooses randomly a bit;

Then we count the number k of 1 bits which have been chosen. This is done in time $O(\mathrm{Log}(n))$ on $O(n)$ processors with the help of the perfect binary tree network

with $\lceil 2n/3 \rceil$ leaves:

Example. If $t = 8$, we use the network shown in Fig. 10.

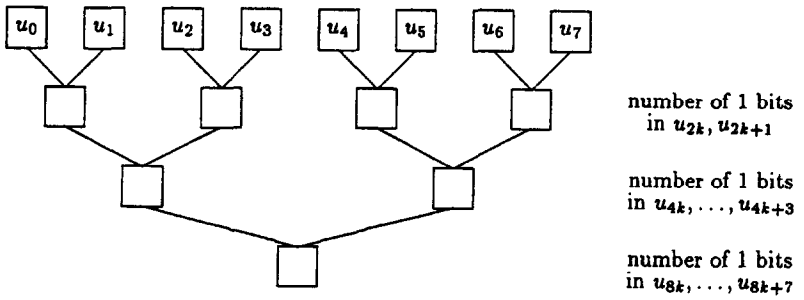


Fig. 10

Then we must verify that the value is accepted or rejected. This is clear if $k = 0$ or when $k > \lfloor (n+1)/2 \rfloor$.

In the other case, we compute the corresponding integers a_k^n, b_k^n and c_k^n in time O(1) with a single processor. Then c_k^n processors are informed that they have to participate to this verification (a classical binary tree structure is used here). This needs $O(c_k^n)$ processors and a time in $O(\mathrm{Log}(c_k^n))$. The ith processor chooses randomly a number N in $[\![1, b_k^n + 1 - i]\!]$ and considers that its drawing succeeds if $N \leqslant a_k^n + 1 - i$. The global result is then recovered, thanks to the tree structure which does now the "*and*" of the values given by the c_k^n processors in time $O(\mathrm{Log}(c_k^n)) = O(\mathrm{Log}(n))$ with $O(c_k^n) = O(n)$ processors.

In case of failure, we start once again the same step (we can inform the $\lceil 2n/3 \rceil$ processors at the beginning in time $O(\mathrm{Log}(n))$, thanks to a binary tree structure with $O(n)$ processors). In case of success, we execute the next step. An attempt for choosing the integer k requires therefore a time in $O(\mathrm{Log}(n))$ and $O(n)$ processors. The total average time for executing this step is therefore also in $O(\mathrm{Log}(n))$.

The second step generates a Motzkin word of size $n - 1$ with $k - 1$ letters x. This is done in time $O(\mathrm{Log}^2(n))$ on $O(n)$ processors using similar methods as for the generation of Dyck words.

4.2. Generation of a left factor of Dyck or Motzkin words

Definition 5. m is a left factor of Dyck (resp. Motzkin) words if and only if there exists a word m' such that mm' is a Dyck (resp. Motzkin) word.

4.2.1. Generation of a left factor of Dyck words

We use the bijection between the left factors of Dyck word with n letters and the sequences of $\lceil n/2 \rceil$ letters x and $\lfloor n/2 \rfloor$ letters y.

We generate a permutation of size n. Then we replace the values of this permutation which are strictly greater than $\lceil n/2 \rceil$ by y and the other ones by x. Now we compute the height $f(0,i)$ of each letter as for the generation of a Dyck word.

Finally, we transform a letter $u_i = x$ (resp. $u_i = y$) into y (resp. x) if and only if $f(0,i) < 0$.

4.2.2. Generation of a left factor of Motzkin words

We proceed in the same way as for the generation of a Motzkin word. First we choose k the number of letters a of our left factor of Motzkin words. Then we generate a left factor with k letters a. (A complete description of the sequential version of this algorithm can be found in [12].)

To find the number of letters a:

(i) First, we generate $\lceil 2n/3 \rceil$ random bits (0 or 1) and call k' the number of 1 bits. Then we generate another random bit x.

(ii) We accept or reject the choices k' and x, depending on a random outcome as we have done for a Motzkin word.

- If $k' < x$ or $k' > \lfloor (n-x)/2 \rfloor$, we reject these choices.
- If $x \leqslant k' \leqslant \lfloor n/3 \rfloor$, we accept these choices with probability $\binom{a^n_{k',x}}{c^n_{k',x}} \Big/ \binom{b^n_{k',x}}{c^n_{k',x}}$ where $a^n_{k',x} = \lfloor n/3 \rfloor$, $b^n_{k',x} = n - x - 2k'$, $c^n_{k',x} = \lfloor n/3 \rfloor - k'$.
- If $\lfloor (n-x)/2 \rfloor \geqslant k' > \lfloor n/3 \rfloor$, we accept the choice of k' with probability $\binom{a^n_{k',x}}{c^n_{k',x}} \Big/ \binom{b^n_{k',x}}{c^n_{k',x}}$ where $a^n_{k',x} = \lceil 2n/3 \rceil - k' - x$, $b^n_{k',x} = k'$, $c^n_{k',x} = k' - \lfloor n/3 \rfloor$.

If the choice of the pair (k',x) is rejected, we go back to the beginning.

Finally, we must generate a left factor of Motzkin words with $k = n - 2k' + x$ letters a. We proceed as follows. First we generate a permutation of n elements. Then we replace the values of this permutation which are in $[\![1,k]\!]$ by a, those which are in $[\![k+1, k+1+\lfloor (n-k)/2 \rfloor]\!]$ by y and the remaining values by x. Finally, we compute the values of $f(0,i)$ and transform a letter $u_i = x$ (resp. $u_i = y$) into y (resp. x) if and only if $f(0,i) < 0$.

5. Conclusion

We have shown how to generate a permutation of size n, a Dyck word, a left factor of Dyck words, a Motzkin word, a left factor of Motzkin words with $O(n)$ processors in time $O(\mathrm{Log}^2(n))$(this is the average complexity of all these generation algorithms and the worst case complexity of the algorithms for generating a permutation, a Dyck word and a left factor of Dyck words).

The same technique applies also to the generation of a word which is in $1-1$ correspondence with a tree split into patterns as defined in [13] (see [3]), but unfortunately

it would be very difficult to use this approach to parallelize the sequential algorithm presented in [15] which generates much more general structures.

Acknowledgements

The authors are grateful to P.Flajolet, D.Gouyou-Beauchamps and J.-G.Penaud for helpful comments.

References

[1] M. Ajtai, J. Komlós and E. Szemerédi, An O(n Log(n)) sorting network, in: *Proc. 15th Ann. ACM Symp. on Theory of Computing* (1983) 1–9.

[2] S.G. Akl and I. Stojmenovic, Generating binary trees in parallel, Report TR-91-22, Université d'Ottawa, 1991.

[3] L. Alonso, Structures arborescentes: algorithmes de génération, problème de l'inclusion, relations maximin, Thèse, Université de Paris-Sud, Centre d'Orsay, 1992.

[4] L. Alonso, Uniform generation of a Motzkin word, *Theoret. Comput. Sci.* **134** (1994) 529–536.

[5] R. Anderson, Parallel algorithms for generating random permutations on a shared memory machine, in: *Proc. SPAA'90* (1990) 95–102.

[6] E. Barcucci, R. Pinzani and R. Sprugnoli, The random generation of directed animals, *Theoret. Comput. Sci.* **127** (1994) 333–350.

[7] K.E. Batcher, Sorting networks and their applications, in: *1968 Spring Joint Computer Conf., AFIPS Proc.*, Vol. 32, Washington, DC (Thompson, 1968) 307–314.

[8] B. Chan and S.G. Akl, Generating combinations in parallel, *BIT* **26** (1986) 2–6.

[9] G.H. Chen and M.S. Chern, Parallel generation of permutations and combinaisons, *BIT* **26** (1986) 277–283.

[10] L. Comtet, Analyse Combinatoire, Presses Universitaires de France, 1970.

[11] T. Cormen, C. Leiserson and R. Rivest, *Introduction to ALGORITHMS* (MIT Press, Cambridge, England, 1990).

[12] A. Denise, Méthodes de génération d'objects combinatoires de grande taille et problème d'énumération, Thèse, Université de Bordeaux I, 1994.

[13] N. Dershowitz and S. Zaks, Patterns in trees, *Discrete Appl. Math.* **25** (1989) 241–255.

[14] N. Dershowitz and S. Zaks, The cycle lemma and some applications, *European J. Combin.* **11** (1990) 35–40.

[15] P. Flajolet, P. Zimmermann and B. V. Cutsem, A calculus for the random generation of combinatorial structures, *Theoret. Comput. Sci.* **132** (1994) 1–35. Also available as Inria Research Report 1830 (anonymous ftp on ftp.inria.fr dir INRIA/publication/RR file RR-1830.ps.gz).

[16] J-G. Penaud, Arbres et animaux, mémoire d'habilitation à diriger les recherches, Université de Bordeaux 1, 1990.

[17] J.L. Rémy, Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire, *RAIRO Inform. Théor.* **19** (2) (1985) 179–195.

[18] H.S. Stone, Parallel processing with the perfect shuffle, *IEEE Trans. Comput.* **c-20** (2) (1971) 153–161.

[19] J. Van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. 1 (Elsevier, Amsterdam, 1990) 886–894.

[20] X.G. Viennot, Combinatoire énumérative, Notes de cours ENS Ulm, Paris, 1989.