Holonomic equations and linear random generation of binary trees

Pierre Lescanne

École Normale Supérieure de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL),
46 allée d'Italie, 69364 Lyon, France
`pierre.lescanne@ens-lyon.fr`
orcid : 0000-0001-9512-5276

Abstract

Holonomic equations are recursive equations that allow computing efficiently numbers of combinatoric objects. Rémy showed that the holonomic equation associated with binary trees yields an efficient linear random generator of binary trees. I extend this paradigm to Motzkin trees and Schröder trees and show that despite slight differences, the algorithms are still linear.

Keywords: combinatorics, random generation, Motzkin number, Catalan number, binary tree, unary-binary tree

1 Introduction

Considering a recurrence defining a sequence of integer coefficients F_n. In this paper, I am interested in specific recurrences called "holonomic recurrence" where roughly speaking, "holonomic" means that F_{n+s} is a combination, using polynomials in n, of the F_i's, for $n \leq i \leq n+s$. More precisely, (see Flajolet and Sedgewick's book [12], Appendix B.4) the coefficients fulfill the following recurrence:

$$P_s(n)F_{n+s} + P_{s-1}(n)F_{n+s-1} + \ldots + P_0(n)F_n = 0$$

for some $n \geq n_0$, where the $P_j(n)$ are polynomials in n. This kind of recurrence is called a *P-recurrence*. For instance, for Catalan numbers:

$$C_n - \sum_{k=0}^{n-1} C_k C_{n-k-1} = 0$$

is the classical recurrence that is used in general to defined them, but it is not a P-recurrence, whereas

$$(n+1)C_n - 2(2n-1)C_{n-1} = 0$$

is the P-recurrence, which will be considered later on.

In this paper, I consider three families of binary trees (binary trees, Motzkin trees aka unary-binary trees, Schröder trees) and their random generation. It turns out that holonomic recurrences play a key role in the design of efficient linear random generation algorithms.

The three examples: binary trees, Motzkin trees, Schröder trees are interesting because they have different holonomic equations, one (Catalan numbers) has one term on the right, one (Motzkin numbers) has a sum of two terms on the right and one (Schröder numbers) has a subtraction of two terms, one the right. These yield different algorithms, as this will be explained further in this paper.

2 Random binary trees

Rémy's algorithm [21] for generation of random binary trees is linear. It is based on a constructive proof of the holonomic equation:

$$(n+1)C_n = 2(2n-1)C_{n-1}$$

Here this holonomic equation is very peculiar since C_n times a polynomial in n is equal to C_{n-1} times a polynomial in n. We will see that this is not the case for Motzkin numbers and Schröder numbers. Rémy's algorithm is described by Knuth in [18] § 7.2.1.6 (pp. 18-19) and works on *extended binary trees*, or just *binary trees* in which we distinguish *internal nodes* and *external nodes* or *leaves*. The idea of the algorithm is that a random binary tree can be built by iteratively and randomly drawing an internal node or a leaf in a random binary tree and inserting, between it and its parent a new internal node and a new leaf either on the left or on the right (see Figure 1). An insertion is also possible at the root. In this case, the new inserted node becomes the root. This is not a specific case in the algorithm as we will see. The root can be seen as the child of an hypothetical node

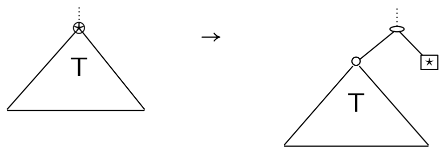
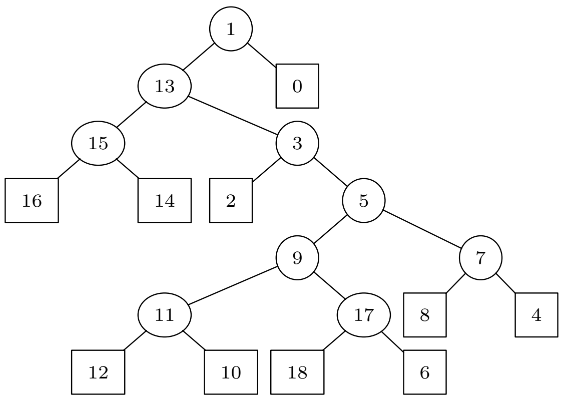


Figure 1: Rémy's right insertion of a leaf

A binary tree of size n has $n-1$ internal nodes and n leaves. We label binary trees with numbers between 0 and $2n-2$ such that internal nodes are labeled with odd numbers and leaves are labeled with even numbers. Inserting a node in a binary tree of size n requires drawing randomly a number between 0 and $4n-3$. This process can be optimized by representing a binary tree as a list (a `vector` in Haskell), an idea sketched by Rémy and described by Knuth. In this vector, even values are for internal nodes and odd values are for leaves. The root is located at index 0. The left child of an internal node with label $2k+1$ is located at index $2k+1$ and its right child is located at index $2k+2$. Here is a vector representing a binary tree with 10 leaves and its drawing.

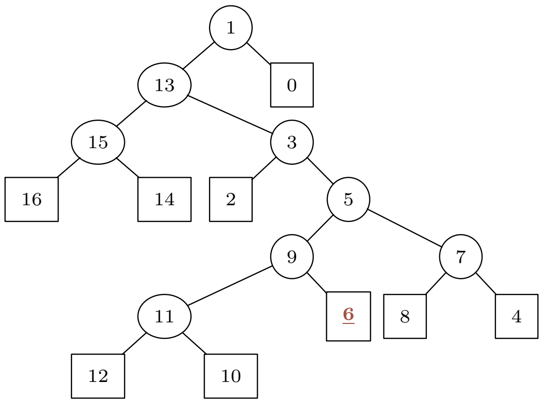
indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
values	1	13	0	2	5	9	7	8	4	11	17	12	10	15	3	16	14	18	6



This tree was built by inserting the node 17 together with the leaf 18 in the following vector.

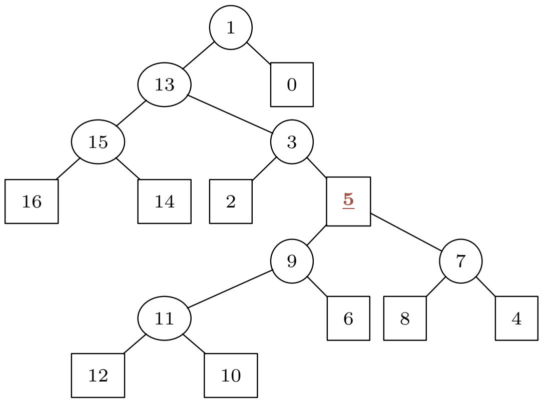
indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
values	1	13	0	2	5	9	7	8	4	11	6	12	10	15	3	16	14

which codes the tree

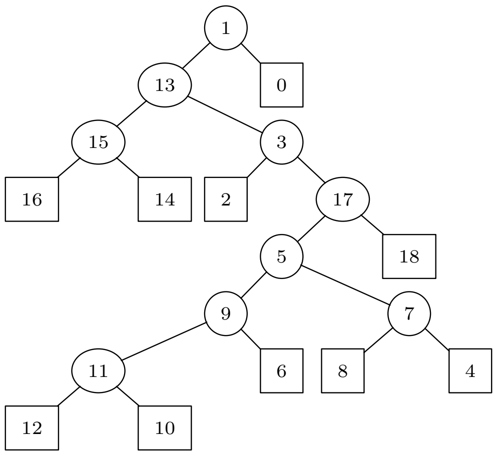


This was done by drawing a node (internal node or leaf, here the node with label 6, right child of the node with label 9) and a direction (here right) and by inserting above this node a new internal node (labeled 17) and, below the new inserted internal node, a new leaf of the left (labeled 18). This double action (inserting the internal node and attaching the leaf) is done by choosing a number in the interval $[0..33]$ (in general, in the interval $[0..(4n-3)]$). Assume that in this case the random generator returns 21. 21 contains two pieces of information : its parity (a boolean) and floor of its half. Half of 21 is 10, which tells that the new node 17 must be inserted above the 11^{th} node (in the vector) namely 6. Since 21 is odd, the rest of the tree (here reduced to the leaf 6) is inserted on the right (otherwise it would be inserted on the left). A new leaf 18 is inserted on the left (otherwise it would be inserted on the right).

Consider the same tree and suppose that the random value is 8. Half of 8 is 4. Hence the new internal node labeled by 17 is inserted above the node labeled by 5



and, since 8 is even, the rest of the tree is inserted on the left and a new leaf (labeled 18) is inserted on the right.



The algorithm (Figure 2) works as follows. If $n = 0$, Rémy's algorithm returns the vector starting at 0 and filled with anything, since the whole algorithm works on the same vector with the same size. In general, say that, for $n-1$, Rémy's algorithm returns a vector v (`vector` is the concept used in Haskell for arrays that can be changed in place). In our Haskell implementation the function yields an object of type `Gen (Vector Int)` which returns vector and carries a hidden random number generator. One accesses to the generator by `get` and stores the new generator by `put`. One draws a random integer x between 0 and $4n-3$. Let k be half of x. In the vector v one replaces the k^{th} position with $2n-1$ and one appends two elements, namely the k^{th} item of v followed by $2n$ if x is even and $2n$ followed by the k^{th} item of v if x is odd.

The algorithm builds a uniformly random *decorated binary tree*, i.e., a binary tree with its leaves numbered 0, 2,... $2n$. We notice that the construction of a tree with such labels is unique, the labels of the internal nodes are a consequence of the construction, hence are deduced from the labels of the leaves. If we ignore the leaves, we get a uniform distribution for the undecorated binary trees (i.e., with no labels on the leaves).

In the program, `rands` is a vector of random floating numbers between 0 and 1.

```
rbt :: Int -> Int -> Gen (Vector Int)
rbt seed 0 = do put (mkStdGen seed)
                return(initialVector // [(0,0)])
rbt seed n =
  do v <- rbt seed (n-1)
     generator <- get
     let (rand, newGenerator) = randomR (0::Double,1) generator
     put newGenerator
     let x = floor (rand * fromIntegral (4*n-3))
          -- x is a random value between 0 and 4n-3 --
         k = x `div` 2
     case even x of
       True -> return(v // [(k,2*n-1),(2*n-1,v!k),(2*n,2*n)])
       False -> return(v // [(k,2*n-1),(2*n-1,2*n),(2*n,v!k)])
```

Figure 2: Haskell program for Rémy's algorithm

3 Motzkin trees

Motzkin trees are also called *unary-binary trees*. This paper proposes a linear algorithm for random generation of Motzkin trees. The algorithm takes the same paradigm as this of Rémy's linear algorithm for random generation of *binary trees* [21]. Assume n is the size of the trees. Recalk that Rémy's algorithm is based on a bijective proof of the inductive equality:

$$(n+1)C_n = 2(2n-1)C_{n-1}$$

where the C_n's are the Catalan numbers. My algorithm for random generation of Motzkin trees is based on a bijective proof due to Dulucq and Penaud [11] of the inductive equality:

$$(n+2)M_n = (2n+1)M_{n-1} + 3(n-1)M_{n-2}$$

where the M_n's are the Motzkin numbers. At some points of the algorithm, choices have to be made, based on M_n and M_{n-1}. Since M_n and M_{n-1} are big numbers, this induces potentially expensive computations on big numbers. To avoid an excessive cost for these steps, I propose a preprocessing. This paper is associated with a Haskell program available on GitHub which serves as an executable specification.

4 Motzkin numbers and Motzkin trees

The n^{th} Motzkin number M_n is the number of different ways of drawing non-intersecting chords between n points on a circle (not necessarily touching every point by a chord). Motzkin numbers count also well parenthesized expressions with a constant **c**, called *Motzkin words*. They are words of length n in the language generated by the grammar M.

$$M \;=\; \varepsilon \mid \mathbf{c}\, M \mid (M)\, M$$

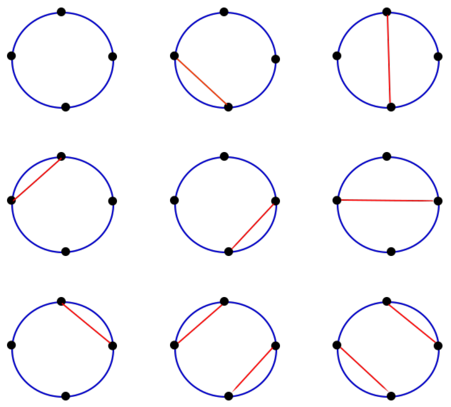
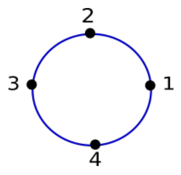


Figure 3: The 9 non-intersecting chords between 4 points on a circle

cccc	**cc**()	**c**(**c**)
c()**c**	(**cc**)	(**c**)**c**
()**cc**	(())	()()

Figure 4: The 9 parenthesized words with constant **c**.

The bijection between sets of non intersecting chords and well parenthesized words with constant **c** is as follows: first one numbers nodes on the circle counterclockwise, as follows:



A position at the beginning of a chord on the circle corresponds to an opening parenthesis. A position at the end of a chord on the circle corresponds to a closing parenthesis. A position which is neither of those corresponds to the constant **c**.

Motzkin numbers count also routes in the upper quadrant from $(0,0)$ to $(0,4)$ with move *up*, *down* and *straight*.

The bijection is as follows: an opening parenthesis corresponds to an *up*, a closing parenthesis corresponds to a *down* and the constant **c** corresponds to a *straight*.

Motzkin number M_n counts also the number in unary-binary planar trees with n edges, that are tree structures with nodes of arity one or two and with n edges. Let us call this number n of edges the *size* of the Motzkin tree. Notice that the number of nodes of a Motzkin tee of size n is $n+1$, i.e., the size plus one. Figure 6 gives the trees for $n = 4$. The bijection f from well parenthesized expressions with constant **c** to Motzkin trees is as follows. Its inverse f^{-1} is also

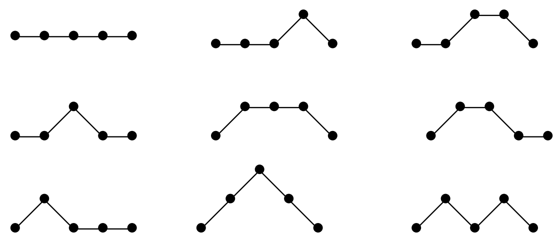
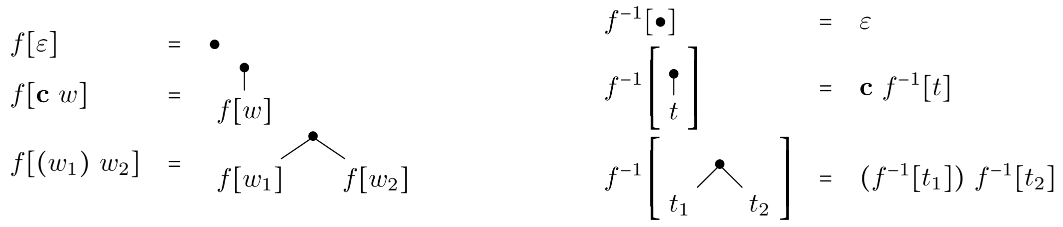


Figure 5: The 9 routes in the upper right quadrant from $(0,0)$ to $(0,4)$ with move *up, down and straight*

given.


$f[\varepsilon] = \bullet$
$f[\mathbf{c}\ w] =$
$f[w]$
$f[(w_1)\ w_2] =$
$f[w_1]$
$f[w_2]$
$f^{-1}[\bullet] = \varepsilon$
f^{-1}
t
$= \mathbf{c}\ f^{-1}[t]$
f^{-1}
t_1
t_2
$= (f^{-1}[t_1])\ f^{-1}[t_2]$

Motzkin numbers fulfill the equation:

$$M_{n+1} = M_n + \sum_{i=0}^{n-1} M_i\, M_{n-1-i}$$

5 Dulucq-Penaud bijection proof

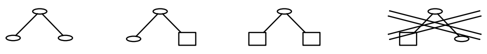
Motzkin numbers fulfill also the *holonomic equation* [12]:

$$(n+2)M_n = (2n+1)M_{n-1} + 3(n-1)M_{n-2}.$$

Together with the equalities $M_0 = 1$ and $M_1 = 1$, Motzkin numbers can be computed and form the sequence A001006 in the *online encyclopedia of integer sequences* [16]. In this section, I present Dulucq and Penaud's proof of this equality [11]. This proof relies on the exhibition of a bijection between the objects counted by the left-hand side and those counted by the right-hand side. The first idea is to consider specific binary trees called *slanting binary trees* and divide those trees into 7 subclasses.

5.1 Slanting binary trees

Following Dulucq and Penaud, I represent *Motzkin trees* as specific *binary trees* in which leaves □ are added. In such binary trees, only the three first configurations below are allowed and the rightmost one is not



The first configuration corresponds to a binary node, the second configuration corresponds to a unary node and the third configuration corresponds to an end node in the classical presentation (for instance in Figure 6). I call such trees *slanting trees.*

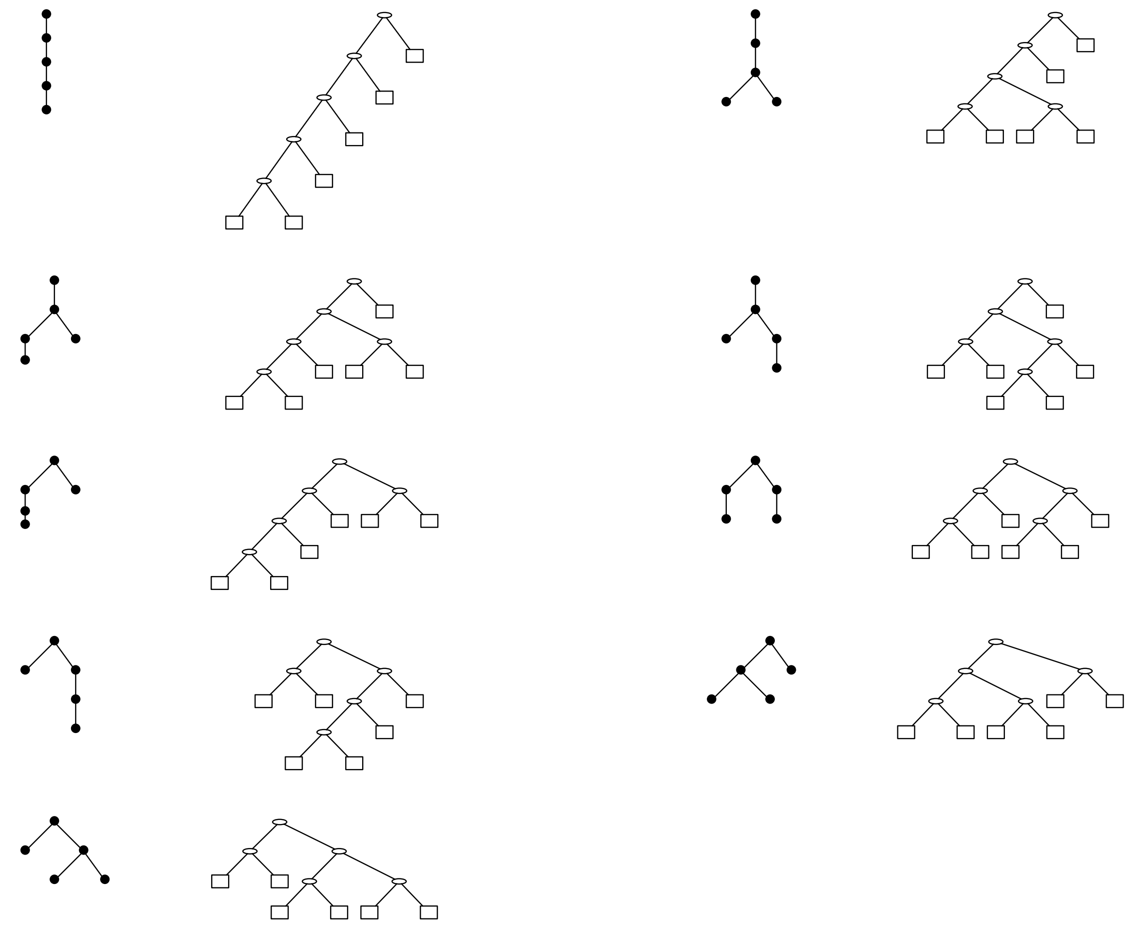
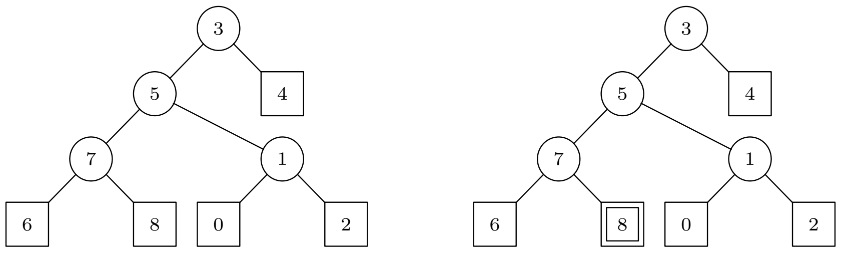


Figure 6: 9 Motzkin trees with 4 edges and their slanting trees

Figure 6 shows the 9 Motzkin trees with 4 edges and their corresponding slanting trees. Let us label each node of a slanting tree with a number between 1 and $2m + 1$, where m is the number of internal nodes of the slanting tree. Let us call such a labeled tree a *labeled slanting tree*. Now consider labeled slanting trees with one marked leaf. Let us call it a *leaf-marked slanting tree*. Below there is a labeled slanting tree of size 4 and a leaf-marked labeled slanting tree, where the mark is on the leaf labeled 8.



This corresponds to the vector:

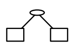
$$[3, 0, 2, 5, 4, 7, 1, 6, 8]$$

How nodes and leaves are labeled by numbers will be explained below and is essentially like binary trees. Just notice that *internal nodes* have odd labels and *leaves* have even labels. From now on, let us forget the labels, but let us mark one of the leaves. In such trees with a marked leaf, we can distinguish 7 general patterns of subtrees containing the marked leaf (Figure 7 first column). The marked leaf is denoted by a star in a square, namely $\boxed{\star}$. In the first group of 4 there are the patterns where the marked leaf is a right child and in the second group of 3, there are the patterns where the marked leaf is a left child, hence, due to the constraints on slanting trees, the other child (a right child) is also a leaf.

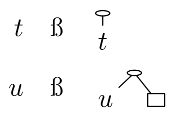
Let us call *node-marked* a slanting tree in which one internal node is marked. Let us call *marked* tree, a slanting tree in which either a leaf or an internal node is marked.

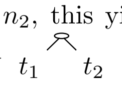
How many leaves in a Motzkin tree?

The slanting tree associated with a Motzkin tree of size n (number of its edges) has $n+2$ leaves. This can be shown by induction.

Basic case: If the Motzkin tree is $\bullet$, its size is 0 and its associated slanting tree  has 2 leaves.

Adding a unary node: Assume we add a unary node above a Motzkin tree t of size n, this yields a Motzkin tree t' of size $n+1$. The slanting tree associated with t' has $n+2$ leaves (the number of the leave of the slanting tree u associated with t) plus a new one added, then all together $n+3$.



Adding a binary node: Assume we add a binary node above two Motzkin trees t_1 and t_2 of size n_1 and n_2, this yields a Motzkin tree t' of size $n_1 + n_2 + 2$. The slanting trees associated with  have $n_1 + n_1 + 4$ leaves.

5.2 A taxonomy of slanting trees

Recall Rémy's algorithm which consists in inserting, at a marked position (internal node or leaf), a leaf in a marked tree (see Figure 1). After insertion, the formerly marked node becomes unmarked and the inserted leaf becomes marked. Here, since we are interested in Motzkin trees, we insert a leaf on the right, above the marked node in the marked slanting tree and like in *Rémy's algorithm*, a leaf insertion on a marked tree is performed, but unlike *Rémy's algorithm* the insertion is performed only on the right and a leaf-marked slanting tree is produced. This corresponds to what is done to *pattern1*, *pattern3* and *pattern4* in the middle column of Figure 7. The leaf is inserted at a marked position in the marked tree of the middle column producing the leaf-marked tree of the left column. But as we will see for the other patterns, there are other ways to increase a slanting tree when it does not correspond to one of these three patterns.

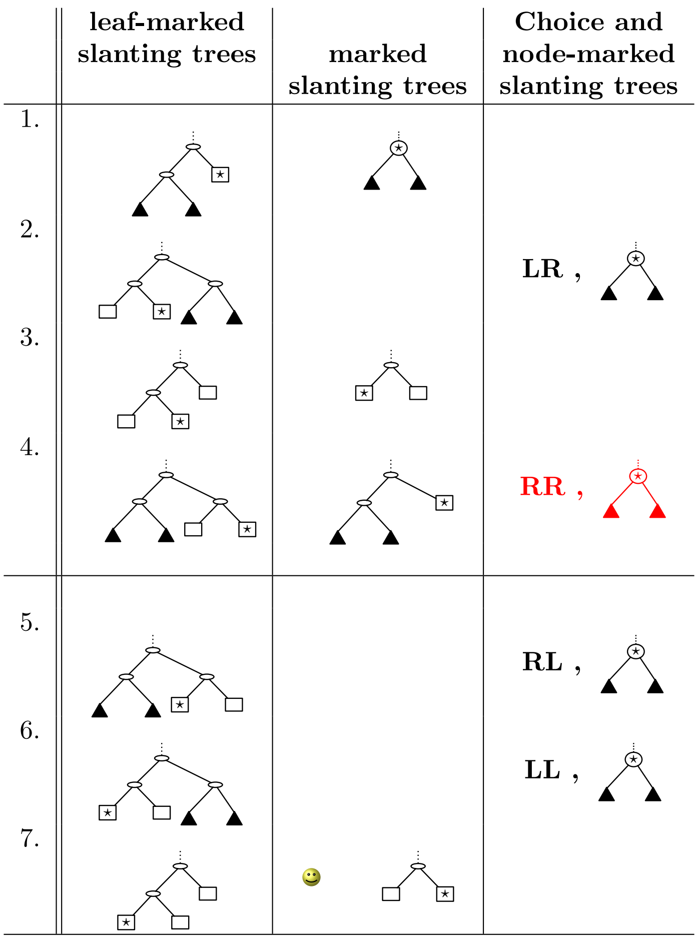

leaf-marked slanting trees
marked slanting trees
Choice and node-marked slanting trees
1.
2.
3.
4.
5.
6.
7.
LR ,
RR ,
RL ,
LL ,

Figure 7: The 7 patterns of leaf-marked slanting trees

6 The bijection

What makes the random generation of Motzkin trees trickier than Rémy's algorithm is the structure of the holonomic equation:

$$(n+2)M_n = (2n+1)M_{n-1} + 3(n-1)M_{n-2}$$

when compared to the equation:

$$(n+1)C_n = 2(2n-1)C_{n-1}$$

First, if we use a construction based on that equation, a Motzkin tree of size n can be built from a Motzkin tree of size $n-1$ or from a Motzkin tree of size $n-2$. Thus there are at least two cases to consider. Actually 6 cases as we will see, since the construction of a Motzkin tree of size $n-1$ splits in 4 cases and the construction of a Motzkin tree of size $n-2$ splits in 3 cases. Notice that M_n counts both the number of Motzkin trees of size n and the slanting trees with $n+2$ leaves.

Interpreting the holonomic equation

We conclude that $(n+2)M_n$ counts the number of *leaf-marked* slanting trees of size n, that $(2n+1)M_{n-1}$ counts the number of *marked* slanting trees of size $n-1$ and that $(n-1)M_{n-2}$ counts the number of *node-marked* slanting trees of size $n-2$. Therefore looking at the equation, we see that we should be able to build a leaf-marked slanting tree of size n from either a marked slanting tree of size $n-1$ or from a pair made of an item which can take one of three values and of a *node-marked* slanting tree of size $n-2$. Let us see how Dulucq and Penaud propose to proceed.

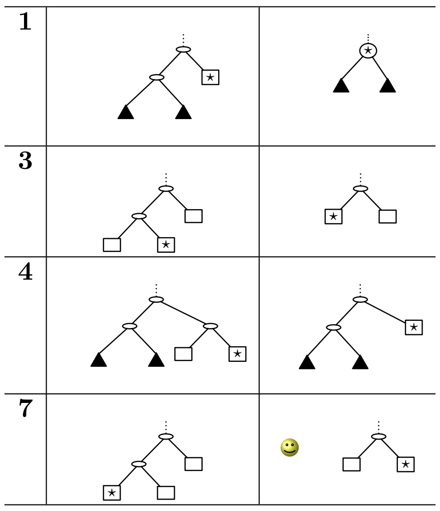


Figure 8: Contribution of marked slanting tree of size $n-1$. 🙂 for *pattern7* marks a specific case explained in Section *The lower part*

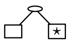
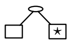
A taxonomy of leaf-marked slanting trees

Leaf-marked slanting trees can be sorted according to the position of their mark. This is done in the first column of Figure 7. This column has two parts.

The upper part

In the upper part, we have four patterns in which the marked leaf is a right child. Let us call them *pattern1*, *pattern2*, *pattern3* and *pattern4*. Three of them *pattern1*, *pattern3* and *pattern4* are obtained by Rémy's right insertion of a leaf (Figure 1) in a marked slanting tree. The other *pattern2* (Figure 9) is not. Indeed if the marked leaf is removed, the tree that is obtained has a leaf on the left and a node on the right, which is forbidden. This pattern will be obtained another way.

The lower part

In the lower part, there are three patterns which correspond to the case where the marked leaf is a left child, hence the sibling of a leaf (Figure 9); *pattern7* cannot be obtained by a right insertion of a leaf (Figure 1), this is why I mark it by 🙂. But Dulucq and Penaud noticed that pattern  among marked slanting trees is not taken into consideration. Thus they propose to associate this pattern  with *pattern7*, as shown in Figure 7.

The bijection by cases

Case $(2n+1)M_{n-1}$

The previously explained contribution to leaf-marked slanting trees of size n from marked slanting tree of size $n-1$ is summarized in Figure 8. All the patterns of marked slanting trees are taken into account.

Case $3(n-1)M_{n-2}$

Let us now look at the three remaining patterns: *pattern2*, *pattern5* and *pattern6*; forming the lines of Figure 9. Those three patterns have the same model, namely an internal node with two children: one is an internal node and the other one is an internal node whose children are two leaves, one of which is marked, the other is not. Depending on the position of the marked leaf, we distinguish three cases.

- **LL** corresponds to the case where the marked node is on the left of the top node and on the left of its parent node.
- **LR** corresponds to the case where the marked node is on the left of the top node and on the right of its parent node.
- **RL** corresponds to the case where the marked node is on the right of the top node and on the left of its parent node.

One notices that there is no case **RR**, because this would correspond to *pattern4* considered in the previous section. As a matter of fact, the three cases **LL**, **LR** and **RL** correspond to the multiplicative factor 3 in the holonomic equation.

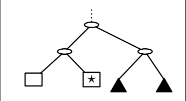
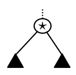
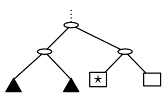
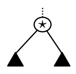
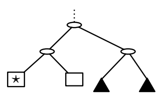
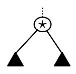
2		**LR**	
5		**RL**	
6		**LL**	

Figure 9: Contribution of node-marked slanting tree of size $n-2$

Forgetting the marks on leaves

As Rémy noticed for binary trees, since we generate the leaf-marked slanting trees of size n uniformly, we also get a uniform distribution of slanting trees of size n. Thus we can forget the marks, which we do in the concrete algorithm.

7 A concrete algorithm for random generation of Motzkin trees

The Haskell program of Figure 19 can be considered as abstract enough to present the algorithm for random generation of Motzkin trees. Thus, in what follows, I make basically no distinction between the algorithm and the program and I consider the Haskell program as an executable specification. The main function is called rMt and returns an object of type Gen (Vector Int) like rbt in Figure 2.

Assume that there is a function motzkin that returns the n^{th} Motzkin number. Like for Rémy's algorithm, one represents a labeled slanting tree by a vector. In this vector, odd labels are for internal nodes and even labels are for leaves. Notice that the algorithm preserves two properties:

1. The vector codes a slanting tree.
2. The vector of a Motzkin tree of size n has a length $2n+3$.

In order to choose which case to consider, namely $(2n+1)M_{n-1}$ (`case1`) or $3(n-1)M_{n-2}$ (`case2`), the algorithm rMt requires a random value between 0 and $(n+2)M_n$ which we call r. If r is less than or equal to $(2n+1)M_{n-1}$, we are in `case1`, else we are in `case2`. Said otherwise, given a random value c between 0 and 1 if $c \leq \frac{(2n+1)Mn-1}{(n+2)M_n}$ we choose `case1`, if not we choose `case2`.

- `case1`: one draws at random a leaf or an internal node in a slanting tree of size $n-1$. This means choosing at random an index k in the vector v. We get *pattern7* if three conditions are fulfilled.

  1. **The marked item, should be a right child**. This means that k **is even**, since the left child of a node of index $2p+1$ is $2p+1$ and the right child of this node is $2p+2$.
  2. **The marked item is a leaf**. This means that $v[k]$ **is even**, since leaves have even labels. Notice that Haskell uses the notation `v!k` for our mathematical notation $v[k]$.
  3. **The sibling item of the marked item is a leaf** (a left child by the way). This means that $v[k-1]$ **is even**.

  In this case (k is even, $v[k]$ is even and $v[k-1]$ is even) one inserts a node and a leaf as shown by Figure 8, which corresponds in the code to:

  ```
  v // [(k-1,2*n+1),(2*n+1,v!(k-1)),(2*n+2,2*n+2)]
  ```

  In Haskell, the operator // updates vectors at once, it is called a bulk update. `(2*n+1,v!(k-1))` means that the left child is a new node, at index $2*n+1$, which points to the former value of `v!(k-1)`. The right child is a new leaf.

  The other cases (*pattern1*, *pattern3*, *pattern4*) correspond to cases when one of k, $v[k]$ or $v[k-1]$ is odd. The update is then

```
    v // [(k,2*n+1),(2*n+1,v!k),(2*n+2,2*n+2)]
```

which corresponds to the first lines of Figure 8.

- `case2`: In this case we consider a random node-marked slanting trees of size $n-2$ and a random values among **LR**, **RL**, **LL**. For that we draw a number r between 0 and $3n-6$, from which $r \div 3$ gives a random number between 0 and $n-2$ (a random node) and r `mode` 3 yields a random number among 0, 1 and 2. We notice that **LR** and **LL** correspond to the same transformation, while **RL** corresponds to another transformation. In each case one adds four nodes, with labels $2n-1$, $2n$, $2n+1$ and $2n+2$. Thus,

```
v // [(2*k+1,2*n-1),(2*k+2,2*n+1),(2*n-1,2*n),(2*n,2*n+2),
                                      (2*n+1,v!(2*k+1)),(2*n+2,v!(2*k+2))]
```

  is the transformation for **LR** and **LL** and

```
v // [(2*k+1,2*n-1),(2*k+2,2*n+1),(2*n-1,v!(2*k+1)),(2*n,v!(2*k+2)),
                                      (2*n+1,2*n),(2*n+2,2*n+2)]
```

  corresponds to **RL**. We let the reader check that the code corresponds to the pictures of Figure 9.

8 Linear algorithm

In the above algorithm, choosing between `case1` and `case2` requires computing M_n and M_{n-1}, numbers of order 3^n. Since for each instance of the algorithm, one uses the same values $(2n+1)M_{n-1}/(n+2)M_n$, these values can be precomputed once for all and stored in a vector, hence making the algorithm linear.

Arithmetic complexity

In arithmetic complexity, in which operation $+$, $-$, $*$ and $/$ take time $O(1)$, the table can be precomputed in time $O(n)$. Therefore the full algorithm is linear.

Bit complexity

In bit complexity, the precomputation takes time $O(n^2)$ and then the algorithm itself is linear.

Results

I precomputed the values, using SAGE [8], until $n = 100\,000$ and the HASKELL program (Figure 11) produces random Motzkin trees in times given by the arrays of Figure 10.

9 Schröder trees

In this section, I define Schröder trees and a proof by bijection of the holonomic equation for defining numbers that count Schröder trees, aka *Schröder numbers* or *Schröder-Hipparcus numbers*. From this proof I derive a quasi linear algorithm for random generation of Schröder trees.

size	time
1 000	1.8s
2 000	3.5s
3 000	5.9s
4 000	7s
5 000	8.9s

size	time
6 000	10.9s
7 000	11.7s
8 000	14.6s
9 000	16.3s

size	time
10 000	20s
20 000	36s
30 000	57s
40 000	83s
50 000	99s

size	time
60 000	234s
70 000	261s
80 000	293s
90 000	322s
100 000	408s

Figure 10: Benchmarks

```
rMtFast :: Int -> Int -> Gen (Vector Int)
rMtFast seed 0 = do put (mkStdGen seed)
                    return $ initialVector // [(0,1),(1,0),(2,2)]
rMtFast seed 1 = do put (mkStdGen seed)
                    return $ initialVector // [(0,1),(1,3),(2,0),(3,2),(4,4)]
rMtFast seed n =
   do generator <- get
      let (rand, newGenerator) = randomR (0::Double,1) generator
      put newGenerator
      case ratioM!n <= rand of
        True -> case1Fast seed n
        False -> case2Fast seed n
```

Figure 11: A program for Motzkin tree random generation with processing (vector ratioM). I omit programs for `case1Fast` and `case2Fast` which can be obtained straightforwardly.

9.1 Definition of Schröder trees

We take the definition of Knuth [18] § 7.2.1.6 (pp. 41): *"A* Schröder tree *is a binary tree in which every nonnull right link is colored either white or black"*. We represent *black* with ⟍ and *white* with ⧹ For instance Schröder trees with 3-nodes are given in Figure 12. Schröder trees are counted by numbers S_n, which fulfill the holonomic equations:

$$(n+1)S_{n+1} - 3(2n-1)S_n + (n-2)S_{n-1} = 0$$

that we will use on the form

$$(n+1)S_{n+1} = 3(2n-1)S_n - (n-2)S_{n-1}$$

Notice that Foata and Zeilberger use for the constructive proof:

$$3(2n-1)S_n = (n+1)S_{n+1} + (n-2)S_{n-1} \tag{1}$$

Schröder trees count also ordinary trees with $n+1$ leaves and no node of degree one (see [18] Exercise 66).

9.2 Foata and Zeilberger bijection

In identity (1):

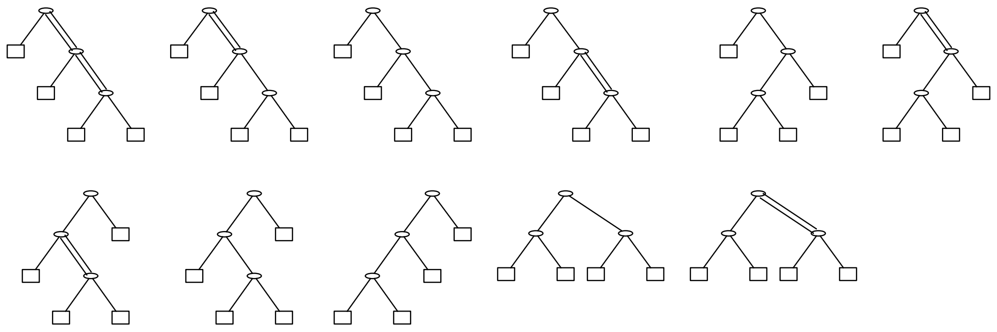


Figure 12: The 11 Schröder trees with 3-nodes.

- $(2n-1)S_n$ counts marked Schröder trees of size n, where nodes and leaves can be marked, therefore $3(2n-1)S_n$, counts pairs made of one of three values and of a marked Schröder tree,
- $(n+1)S_{n+1}$ counts leaf-marked Schröder trees of size $n+1$,
- $(n-2)S_{n-1}$ counts node-marked Schröder trees of size $n-1$.

In order to prove identity (1), Foata and Zeilberger build a one-to-one function which associates, with a pair of a number 0, 1 or 2 (coded by Foata and Zeilberger as L_1, L_2 and R_1) and a marked Schröder tree of size n, either a leaf-marked Schröder tree of size $n+1$ or a node-marked Schröder tree of size $n-1$.

Foata and Zeilberger bijection that proves identity (1) is based on constructions similar to Rémy's insertion (Figure 1), with the difference that there are left insertions corresponding to black or white links (Figure 13). However there are three patterns (Figure 14), with impossible

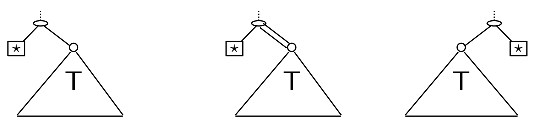


Figure 13: Insertion of a leaf in Foata and Zeilberger bijection

insertion, when one tries to insert a left leaf which has a sibling with a white link on the right (middle case in Figure 13).

Similarly there are two unreachable cases by left or right insertion. They are patterns which are the results of a right or left insertion on a leaf which are a right child by a white link.

Let us call L_1, L_2 and R_1 the three labels. Therefore Figure 16 gives the correspondence between pairs of a label from $\{L_1, L_2, R_2\}$ and of a marked Schröder tree, with either a leaf-marked tree of size $n+1$ or a node marked tree of size $n-1$.

Cases L_1 and R_1. One inserts a leaf with a black link. This is exactly like Rémy's insertion.

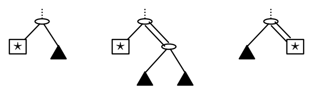


Figure 14: Three impossible left insertion of leaves

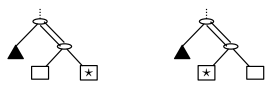


Figure 15: Two unreachable

Case L_2. This case deals with a left insertion with a white right link.

First line: The mark is on a node. The left insertion with a white right link is possible.

Second line: The mark is on a left leaf and the right link is black. The left insertion yields a forbidden pattern (a right white link toward a leaf). But by twisting the tree and swapping the right links, one reach the first unreachable pattern of Figure 15.

Third line: The mark is on a left leaf and the right link is white. Then the left child is not a leaf. The left insertion is forbidden as well. Therefore by removing the left leaf one gets a node marked tree of size $n - 1$.

Fourth line: The mark is on a right leaf. The left insertion yields a forbidden pattern, but by swapping the right link one gets the second unreachable pattern of Figure 15.

10 A concrete algorithm for random generation of Schröder trees

The program of Figure 17 presents the algorithm for random generation of Schröder trees. Like in previous cases one uses an array of size $2n + 1$. But in addition to the indices for the next nodes, one adds a boolean. This boolean says that the link that starts from the node corresponding to this index is white. Hence each cell of the array contains a pair (k, b) where k is an index and b is a boolean. According to the constraints induced on the Schröder trees, the pair corresponding to a boolean $True$ has as a first component an odd number, since this first component corresponds to a link to a node. Since the node is a right child, it is located at an even index. If these constraints are not fulfilled, the second component must be a $False$. Therefore in the program when the programmer adds a pair $(k, True)$ at a position m, she has to check that k is odd and m is even. In another hand, if she writes $(m, (k, True))$ for such an operation, the programmer has to check that it is of the form $(2p, (2q + 1, True))$. This constraints is an invariant of the program. The case when Foata and Zeilberger produce a Schröder tree of size $n - 2$ corresponds in my algorithm to a "failure", that is a "retry": the subprogram `body` is called with a new generator.

There are six cases. Assume one draws a number x between 0 and $6n - 4$ and let us call k the number $x \div 3$. The values $x \mod 3$ discriminate among L_1, L_2 and R_2: 0 for L_1, 1 for

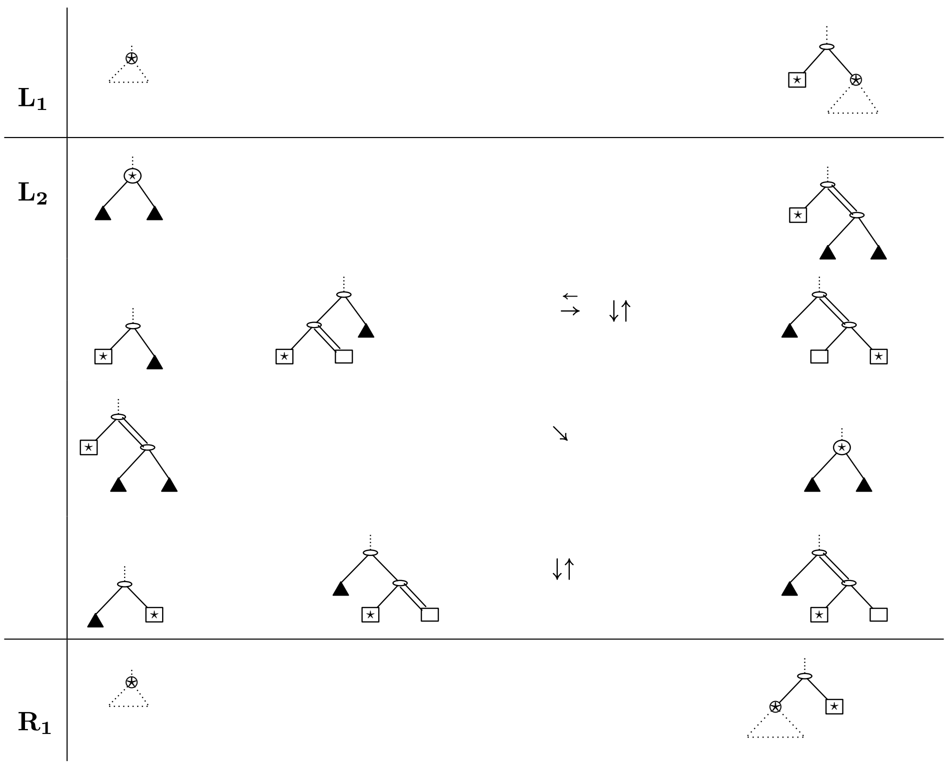


Figure 16: Foata and Zeilberger isomorphism in pictures

L_2 and 2 for R_1. The cases L_1 and R_1 are easy. Assume also that at the k^{th} index the array contains (h, b).

L_1 corresponds in the code to

```
(v // [(k,(2*n-1,False)),(2*n-1,(2*n,False)),(2*n,v!k)]
```

The links are black hence the booleans are *False*.

L_2 **and** h **is odd.** This means that the mark is on a node. This corresponds to the code:

```
v// [(k,(2*n-1,False)),(2*n-1,(2*n,False)),(2*n,(fst(v!k,True)))]
```

Clearly, `2*n,(fst(v!k),True)` fulfills the constraints $(2p, (2q+1, True))$ since `fst(v!k)`, that is h, is odd.

L_2 **and** h **is even and** k **is odd and the other link is black and the second component of `v!(k+1)` is *False*:** This means that the mark is on a leaf (h is even) which is a left child (k is odd) and the link that goes to the sibling leaf is black. This corresponds to the code:

```
type Gen = State StdGen

rst :: Int -> Int -> Gen (Vector (Int,Bool))
rst seed 0 = do put (mkStdGen seed)
                return (initV // [(0,(0,False))])
rst seed n =
  let g = rst seed (n-1)
      body generator =
        do v <- g
           let (rand, newGenerator) = randomR (0::Double,1) generator
               x = floor (rand * fromIntegral (6*n-4))
               -- x is a random value between 0 and 6n - 4
               k = x `div` 3
           put newGenerator
           case x `mod` 3 of
             0 {-L1-} -> return (v // [(k,(2*n-1,False)),
                                       (2*n-1,(2*n,False)),
                                       (2*n,v!k)]) -- L1
             1 {-L2-} -> case odd(fst (v!k)) of
               True {- not a leaf-}-> return (v// [(k,(2*n-1,False)),
                                                    (2*n-1,(2*n,False)),
                                                    (2*n,(fst(v!k),True))])
               False -> case odd k  of
                  True {- the leaf is on the left -} -> case snd (v ! (k+1)) of
                    False {- the other link is black-} ->
                      return (v//[(k,v!(k+1)),
                                  (k+1,(2*n-1,True)),
                                  (2*n-1,v!k),(2*n,(2*n,False))])
        {-Failure-} True {- the other link is white -} -> body newGenerator
                  False {- the leaf is on the right -} ->
                    return (v//[(k,(2*n-1,True)),
                                (2*n-1,v!k),
                                (2*n,(2*n,False))])
             2 {-R1-} ->  return (v // [(k,(2*n-1,False)),
                                        (2*n-1,v!k),
                                        (2*n,(2*n,False))])
  in do generator <- get
        body generator
```

Figure 17: Haskell program for random generation of Schröder trees.

```
v//[(k,v!(k+1)),(k+1,(2*n-1,True)),(2*n-1,v!k),(2*n,(2*n,False))]
```

The reader may check that the code corresponds to the picture of Figure 16, line 2. `(k+1,(2*n-1,True))` fulfills the constraints $(2p,(2q+1,True))$ since $k+1$ is even and $2n-1$ is odd.

L_2 and h is even and k is odd and the other link is white and the second component

of `v!(k+1)` is *True*: This corresponds to a *failure*, and the program loops with a new random generator.

L_2 and h is even and k is even: This means that the mark is a leaf (h is even) which is a right child. This corresponds to the code:

```
v//[(k,(2*n-1,True)),(2*n-1,v!k),(2*n,(2*n,False))]
```

`(k,(2*n-1,True))` fulfills the constraints $(2p,(2q+1,True))$ since k is even and $2n-1$ is odd.

R_2 corresponds in the code to

```
v // [(k,(2*n-1,False)),(2*n-1,v!k),(2*n,(2*n,False))
```

The links are black hence the booleans are $False$.

One may notice that in the array, only value at odd indices require to carry a boolean, and this boolean is required only when the first value of the pair is odd. This suggests a better data structure which may save space.

10.1 Ratio of failures and asymptotic quasi-linearity

The ratio of failures or retries is

$$\frac{(n-2)S_{n-1}}{3(2n-1)S_n} \sim \frac{1}{6(3+\sqrt{8})} \approx 0.0286$$

See [17], exercise 12, §2.2.1. p.534, which gives an asymptotic approximation of Schröder numbers. This means asymptotically that failure occurs in less than 3% of the case. Long sequences of retries are very unlikely. This also means that the algorithm is quasi-linear on the average.

10.2 Benchmarks

Given a size, generation time is slightly sensitive to the seed. In Figure 18, there are benchmarks with seed 0. Until 20 000, computations are made on a laptop (with 4 *Gb* of memory) and further on a workstation (with 400 *Gb* of memory).

size	time
1 500	2.1s
2 000	3.6s
3 000	6.1s
4 000	7.3s
5 000	9.6s

size	time
6 000	12.2s
7 000	12.9s
8 000	15.3s
9 000	19.5s

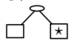
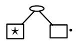
size	time
10 000	20s
20 000	18.2s
30 000	30s
40 000	35.5s
50 000	54.9s

size	time
60 000	140s
70 000	169s
80 000	186s
90 000	247s
100 000	272s

Figure 18: Benchmarks for Schröder trees

11 Related Works

Our work was inspired by Jean-Luc Rémy's algorithm [21]. Laurent Alonso [1] proposed an algorithm for generating uniformly Motzkin trees. His method consists in generating the number k of binary nodes with the correct probability law; he uses then standard techniques to generate a unary-binary tree with k binary nodes among n nodes. The number of trees with k binary nodes is over-approximated by values that follow a binomial distribution: choosing k is therefore done using random generations for a binomial law and rejections. Therefore his algorithm is *linear on the average*, with possible but rare long sequences of rejection. Dominique Gouyou-Beauchamps and Cyril Nicaud [15] propose a random generation for color Motzkin trees which is linear on the average and Srečko Brlek et al. [6] propose an extension of Alonso's algorithm to generalized versions of Motzkin trees.

Axel Bacher, Olivier Bodini and Alice Jacquot [3] propose an algorithm with similar ideas. Especially their Figure 2 shares similarity with our Figure 9. There "operations" G_3, G_4 and G_5 are connected with our cases **RL**, **LR** and **LL** respectively. ∅ corresponds to  and ⦸ corresponds to . However the algorithm they propose has a linear expected complexity, like Alonso's. Let us also mention generic Boltzmann's samplers with exact-size which apply among others to Motzkin trees [5, 19] and generic algorithms [9, 20] with linear expected complexity.

Denise and Zimmermann [7] discuss what can be done on floating-point arithmetic when generating random structures. The authors focus on decomposable labeled structures [13] and address the problem of choice, with a specific section on Motzkin trees and a non linear algorithm.

After studying the random generation of Motzkin trees and reading Foata and Zeilberger paper, I started the implementation of a random generation of Schröder trees, which turns out to be quasi-linear. Actually Laurent Alonso, René Schott and Jean-Luc Rémy [2] proposed another linear algorithm for random generation of Schröder trees, on the same principle as Alonso's algorithm for the generation of Motzkin trees. Like this quasi-linear algorithm, it proceeds in two steps: first, it chooses randomly the number k of leaves with an adequate probability and a rejection technique, second, it generates a random Schröder tree with k leaves. In comparison, my algorithm is direct. I deal only with Schröder trees, not with Schröder trees with k leaves and I am very closed to Rémy's algorithm and to my algorithm for random generation of Motzkin trees.

Acknowledgments

I thank Laurent Alonso, Maciej Bendkowski, Éric Fusy, Alain Giorgetti and Jean-Luc Rémy for interesting discussions and suggestions.

12 Conclusion

Generating Motzkin trees and Schröder trees has many potential applications [10, 4, 2]. My algorithm for generation of Motzkin trees has a simple code and is linear and my algorithm for Schröder trees is direct. Among the possible extensions of my method which could be explored, there is the generation of extended versions of Motzkin structures like Motzkin trees with colored leaves [15] or Motzkin paths with k long straights [6]. $k = 1$ corresponds to Motzkin paths and $k = 2$ to Schröder paths. On another hand, Bracucci et al. [4] study a family of sets of permutations: $\mathcal{M}_1$, $\mathcal{M}_2$, ..., $\mathcal{M}_\infty$, in which $\mathcal{M}_1$ is for Motzkin permutations (that are Motzkin trees up to a bijection) and $\mathcal{M}_\infty$ is for Catalan permutations (that are binary trees up to a bijection) an interpolation of our method seems doable.

References

[1] Laurent Alonso. Uniform generation of a Motzkin word. *Theor. Comput. Sci.*, 134(2):529–536, 1994.

[2] Laurent Alonso, Jean-Luc Rémy, and René Schott. Uniform generation of a Schröder tree. *Inf. Process. Lett.*, 64(6):305–308, 1997. doi:10.1016/S0020-0190(97)00174-9.

[3] Axel Bacher, Olivier Bodini, and Alice Jacquot. Efficient random sampling of binary and unary-binary trees via holonomic equations. *CoRR*, abs/1401.1140, 2014.

[4] Elena Barcucci, Alberto Del Lungo, Elisa Pergola, and Renzo Pinzani. From Motzkin to Catalan permutations. *Discret. Math.*, 217(1-3):33–49, 2000. doi:10.1016/S0012-365X(99)00254-X.

[5] Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. Polynomial tuning of multiparametric combinatorial samplers. In Markus E. Nebel and Stephan G. Wagner, editors, *Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2018, New Orleans, LA, USA, January 8-9, 2018*, pages 92–106. SIAM, 2018. doi:10.1137/1.9781611975062.9.

[6] Srecko Brlek, Elisa Pergola, and Olivier Roques. Non uniform random generation of generalized Motzkin paths. *Acta Informatica*, 42(8-9):603–616, 2006. doi:10.1007/s00236-006-0008-x.

[7] Alain Denise and Paul Zimmermann. Uniform random generation of decomposable structures using floating-point arithmetic. *Theor. Comput. Sci.*, 218(2):233–248, 1999. doi:10.1016/S0304-3975(98)00323-5.

[8] The Sage Developers, William Stein, David Joyner, David Kohel, John Cremona, and Burçin Eröcal. Sagemath, version 9.0, 2020. URL: http://www.sagemath.org.

[9] Luc Devroye. Simulating size-constrained Galton-Watson trees. *SIAM J. Comput.*, 41(1):1–11, 2012. doi:10.1137/090766632.

[10] R. Donaghey and L. W. Shapiro. Motzkin numbers. *Journal of Combinatorial Theory, Series A*, 23(3):291–301, 1977.

[11] Serge Dulucq and Jean-Guy Penaud. Interprétation bijective d'une récurrence des nombres de Motzkin. *Discret. Math.*, 256(3):671–676, 2002. doi:10.1016/S0012-365X(02)00342-4.

[12] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics.* Cambridge University Press, 2008.

[13] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994. doi:10.1016/0304-3975(94)90226-7.

[14] Dominique Foata and Doron Zeilberger. A classic proof of a recurrence for a very classical sequence. *J. Comb. Theory, Ser. A*, 80(2):380–384, 1997. doi:10.1006/jcta.1997.2814.

[15] Dominique Gouyou-Beauchamps and Cyril Nicaud. Random generation using binomial approximations. In *21st International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA'10)*, pages 359–372, Vienna, Austria, 2010.

[16] OEIS Foundation Inc. The on-line encyclopedia of integer sequences. Published electronically at https://oeis.org/, 2022.

[17] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Company, 1973.

[18] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions.* Addison-Wesley Publishing Company, 2005.

[19] Pierre Lescanne. Boltzmann samplers for random generation of lambda terms. *CoRR*, abs/1404.3875, 2014. URL: http://arxiv.org/abs/1404.3875, arXiv:1404.3875.

[20] Konstantinos Panagiotou, Leon Ramzews, and Benedikt Stufler. Exact-size sampling of enriched trees in linear time, 2021. URL: https://arxiv.org/abs/2110.11472, doi:10.48550/ARXIV.2110.11472.

[21] Jean-Luc Rémy. Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire. *RAIRO Theor. Informatics Appl.*, 19(2):179–195, 1985. doi:10.1051/ita/1985190201791.

```haskell
type Gen = State StdGen

rMt :: Int -> Int -> Gen(Vector Int)
rMt seed 0 = do put (mkStdGen seed)
                return (initialVector // [(0,1),(1,0),(2,2)])
rMt seed 1 = do put (mkStdGen seed)
                return (initialVector // [(0,1),(1,3),(2,0),(3,2),(4,4)])
rMt seed n =
  do generator <- get
     let (rand, newGenerator) = randomR (0::Double,1) generator
         floatToInteger x n = floor (x * (fromIntegral n))
         r = floatToInteger rand ((fromIntegral (n+2))*(motzkin n))
         b = r <= (fromIntegral (2*n+1)) * (motzkin (n-1))
     put newGenerator
     case b of
       True -> case1 seed n
       False -> case2 seed n

-- (even k) means that the marked item is a right child
-- (even (v!k)) means that its label is even, hence it is a leaf
-- (even (v!(k-1)) means that the label of its left sister is even,
--                  hence it is a leaf
-- Therefore {(even k) && (even (v!k) && (even (v!(k-1))} is configuration 7
case1 :: Int -> Int -> Gen (Vector Int)
case1 seed 0 = do return initialVector
case1 seed 1 = do return initialVector -- 0, 1 are never invoqued
case1 seed n =
  do generator <- get
     let (rand, newGenerator) = randomR (0::Double,1) generator
         k = floor (rand * (fromIntegral (2*n)))
     v <- rMt seed (n-1)
     put newGenerator
     case odd k || odd (v!k) || odd (v!(k-1)) of
       True -> return (v // [(k,2*n+1),(2*n+1,v!k),(2*n+2,2*n+2)])
       False -> return (v // [(k-1,2*n+1),(2*n+1,v!(k-1)),(2*n+2,2*n+2)])

case2 :: Int-> Int -> Gen(Vector Int)
case2 seed 0 = do return initialVector
case2 seed 1 = do return initialVector  -- 0, 1 are never invoqued
case2 seed n =
    do generator <- get
       let (rand, newGenerator) = randomR (0::Double,1) generator
           r = floor (rand * (fromIntegral (3*n-6)))
           k = r `div` 3
           c = r `rem` 3
       v <- rMt seed (n-2)
       put newGenerator
       case c < 2 of
         True -> return (v // [(2*k+1,2*n-1),(2*k+2,2*n+1),
                               (2*n-1,2*n),(2*n,2*n+2),
                               (2*n+1,v!(2*k+1)),(2*n+2,v!(2*k+2))])
         False -> return (v // [(2*k+1,2*n-1),(2*k+2,2*n+1),
                                (2*n-1,v!(2*k+1)),(2*n,v!(2*k+2)),
                                (2*n+1,2*n),(2*n+2,2*n+2)])
```

Figure 19: Haskell program for random generation of Motzkin trees