Random Sampling of Ordered Trees according to the Number of Occurrences of a Pattern

Gwendal Collet[a], Julien David[b], Alice Jacquot[b]

[a] *LIX UMR 7161, École Polytechnique, 91128 Palaiseau cedex, France*
[b] *LIPN UMR 7030, Université Paris 13—CNRS, 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse, France* [1]

Abstract

In this paper we describe an algorithm to automatically build the combinatoric specification of ordered trees according to their size and the number of occurrences of a given pattern. This specification can easily be converted into an algebraic system, enabling the use of classical analytic combinatorics methods including the ones used for random sampling. We also give a linear pattern matching algorithm which is an adaptation of the localization automaton on trees.

1. Introduction

Random samplers of combinatorial objects are a powerful tool in applied and theoretical computer science: they can be used to experimentally compare the efficiency of several algorithms solving a given problem; they allow to test or build conjectures on the properties of the combinatorial objects or on the average/generic complexity of an algorithm applied to this object. During the last decades, generic methods have been developed to automatically produce random samplers: Markov chains [15], the recursive method [12], the Boltzmann Sampler [10, 9, 4]. The last two methods both use the generating function of the studied object. Given a set of parameters, they allow to generate objects uniformly amongst those with the same values. Therefore, in order to automatically build samplers that follow this paradigm, it is essential to be able to automatically build the multivariate generating function of the studied object.

In this paper, the studied combinatorial objects are *ordered* (or *plane*) trees. We show how to automatically build generating functions and random samplers in which we can parametrize the exact/average number of occurrences of a given pattern. Hence, the parameters we consider on the objects are their size and the number of occurrences of a pattern. The random sampling of several combinatorial objects weighted according to occurrences of or avoiding patterns have been recently studied (words [1], permutations [2], walks [3]).

[1] This work was partially supported by the ANR MAGNUM project

The algorithm we propose builds a grammar that marks the occurrences of a pattern, from which one can easily deduce a multivariate generating function. A random sampler can automatically be obtained from this function using the recursive method and almost automatically with the Boltzmann method (since there is no algorithm to extract the main singularity of a generating function). The probability distribution is still uniform but on objects sharing the same size and number of occurrences of the pattern. In [6], the authors prove that the distribution of pattern occurrences follows a Gaussian law. They use an algorithm, similar to ours, to build a grammar for unordered trees. Though, since their objective is not to obtain a random sampler, they do not optimize the size of the grammar which, in their case, is always exponential in the height of their patterns. In this paper, the grammar for ordered trees obtained from the algorithm might have a smaller set of non-terminal than theirs. It is still exponential in the worst case scenario (as it will be detailed in the paper), but is linear in the best case.

Our grammar can also be used to obtain a linear pattern matching algorithm. In [14], the author describes a bottom-up and a top-down linear algorithms, based on automata, for tree pattern matching. The grammars associated to those automata seem to be unfit for random generation:

- the top-down algorithm decomposes the tree-pattern into several string-paths and therefore one can not directly mark an occurrence of the pattern in the grammar.
- the bottom-up algorithm would lead to a bottom-up grammar. Though, the recursive method and Boltzmann sampler are currently more adapted to top-down grammar. The grammar we obtain can be seen as a top-down version of this algorithm.

Flouri *et al.* [13] consider a similar problem: the authors present an algorithm which builds an automaton computing the number of occurrences of a given suffix in any ordered tree. Extending their automaton to compute occurrences of a pattern (instead of a suffix) seems to require a more involved machinery than in the case of words, due to the very structure of trees: many occurrences of a pattern can overlap.

The paper is organized as follows. Section 2 is devoted to definitions on trees and patterns. In Section 3, we give an algorithm which, given a pattern, produces a grammar from which one can automatically obtain a multivariate generating function. The algorithm is decomposed in several parts in order to improve readability. Then, we show in Section 4 that using this grammar, we can obtain an efficient random sampler and, in Section 6, a linear pattern matching algorithm. Section 5 is dedicated to experimental results.

2. Definitions

A *combinatorial class* is a (finite or infinite) set E of objects, associated with a size function $|.| : E \to \mathbb{N}$ such as for all $n \in \mathbb{N}$ there is a finite number of objects of size n. A *recursive class* is a combinatorial class defined recursively from an object ε of size 0, called *neutral object* and an object $\mathcal{Z}$ of size 1, called *atom*, with some operators such as union, product... The set of atoms $\mathcal{A}$ of a structure is the set of the occurrences of the atom in the object. The size of a structure is the cardinal of $\mathcal{A}$. An unlabelled class is a class where the objects are seen up to isomorphism upon $\mathcal{A}$. Readers unfamiliar with the recursive description of combinatorial class can see for example [11].

2.1. Trees

Let S be a (finite or infinite) subset of $\mathbb{N}$ with $0 \in S$ of allowed arities. A *tree over* S is a tuple $t = (\mathcal{Z}, t_1, \dots, t_r)$ where $\mathcal{Z}$ is the atom, $r \in S$, and for all i in $\{1, \dots, r\}$, t_i is a tree over S. We call this occurrence of $\mathcal{Z}$ the root node of the tree. The occurrences of the atom in the structure are called nodes. The trees $t_1, \dots, t_r$ are called *children* of t and t is their *parent*. If $r = 0$ then t is called a leaf. Each tree has a linear representation derived from this recursive expression (see Figure 1). The node i is the i-th occurrence of an atom $\mathcal{Z}$ in the linear representation. It is also the i-th node encountered in a depth-first search.

Let t be a tree and i one of its nodes of arity r. The *subtree rooted at* i is the tuple whose first element is the node i.

In particular, $\mathcal{T}$ the combinatorial class of tree over S follows the recurrence:

$$\mathcal{T} = \sum_{a \in S} \mathcal{Z}\mathcal{T}^a$$

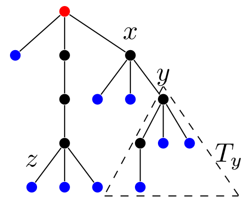


Figure 1: A tree t over $S = \{0, 1, 3, 5\}$. Its root is coloured in red, its leaves in blue. x, y and z are nodes of the tree. The subtree rooted in y is T_y, both x and y are of arity 3: they have 3 children. Its linear representation would be: $\big(\mathcal{Z}, (\mathcal{Z}), \big(\mathcal{Z}, (\mathcal{Z}, (\mathcal{Z}, (\mathcal{Z}), (\mathcal{Z}), (\mathcal{Z})))\big), \big(\mathcal{Z}, (\mathcal{Z}), (\mathcal{Z}), (\mathcal{Z}, (\mathcal{Z}, (\mathcal{Z})), (\mathcal{Z}), (\mathcal{Z}))\big)\big)$ We have $depth_t(T_y) = 2$ and $height(T_y) = 2$.

The *height* and *depth* of a subtree t in a tree p are defined by:

$$\text{height}(t) = \begin{cases} 0 & \text{if } t \text{ if a leaf,} \\ 1 + \max\{\text{height}(t_i) \mid t_i \text{ is a child of } t\} & \text{otherwise.} \end{cases}$$

$$\text{depth}_p(t) = \begin{cases} 0 & \text{if } t = p, \\ 1 + \text{depth}_p(m) \text{where } m \text{ is the parent of } t & \text{otherwise.} \end{cases}$$

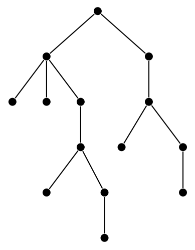
2.2. Prefix, suffix and patterns

A tree s is a *suffix* of t if there exists a node n in t such that s is the subtree rooted in n.

A tree p is a *prefix* of t if there exists $k \in \mathbb{N}$, a tuple $\mathcal{L} = (l_1, \dots, l_k)$ of leaves of p and a tuple of trees $\mathcal{S} = (s_1, \cdots, s_k)$ such that one can obtain t by substituting each leaf l_i by a tree s_i.

Definition 1 (Pattern). *A tree p is a* pattern *of t if there exists a suffix s of t such that p is a prefix of s.*

This definition of pattern should not be mistaken with the notion of subtree in the literature, which corresponds to what we call a suffix.



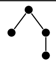
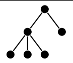
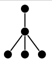
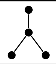
Tree	Occurences as a prefix	Occurrences as a suffix	Occurences as a pattern
	1	2	3
	1	0	1
	0	0	0
	0	0	2

Figure 2: Examples of prefixes, suffixes and patterns: for given patterns, the tabular on the right show how many times they appears as prefix, suffix or pattern in the tree on the left.

The following definitions generalizes the one in [7], since we consider forests instead of trees.

Definition 2 (Context and buds). *Let F be an ordered forest (a tuple of trees) and $(n_1, \ldots, n_\ell)$ a tuple of leaves in F. The pair $C = \{F, (n_1, \ldots, n_\ell)\}$ is called a* context*. The leaves $(n_1, \ldots, n_\ell)$ are called the buds of C. We note $buds(C) = (n_1, \ldots, n_\ell)$.*

Definition 3 (Substitution). *Let $C_1 = \{F_1, (n_1, \ldots, n_\ell)\}$ and $C_2 = \{F_2, (n'_1, \ldots, n'_{\ell'})\}$ be two contexts where F_2 is an ℓ-forest. We denote $C_1[C_2]$ the context $\{F_3, (n'_1, \ldots, n'_{\ell'})\}$ where F_3 is obtained by replacing each n_i in F_1 by the tree t_i in F_2.*

Definition 4 (Stump). *The* stump *of a tree t at depth d (with $0 \leq d \leq height(t)$), noted $stump_t(d)$, is a context $\{F, (n_1, \ldots, n_\ell)\}$ where F is only a tree, the truncation of t at depth d, i.e., the tree reduced to the nodes of depth at most d. The tuple of leaves $(n_1, \ldots, n_\ell)$ are leaves of F whose arity is strictly greater than 0 in t. Note that there are exactly* $\mathrm{height}(t) + 1$ *stumps of t.*

Definition 5 (Segment). *A segment is a context $\{s, (n_1, \ldots, n_\ell)\}$, where $s \in S^{\mathbb{N}}$ is tuple of trees of depth at most 1. The* segment *of a tree t of parameter $d \geq 1$, noted $segment_t(d)$, is the context composed of all the internal nodes at depth d and all nodes at depth $d+1$. The buds of $segment_t(d)$ is the set of internal nodes of t at depth $d+1$. We have $stump_t(d) = stump_t(d-1)[segment_t(d-1)]$. More generally, a segment s is said to be* compatible *with a context C if $|s| = |buds(C)|$.*

Figure 3 shows an example of stumps and segments.

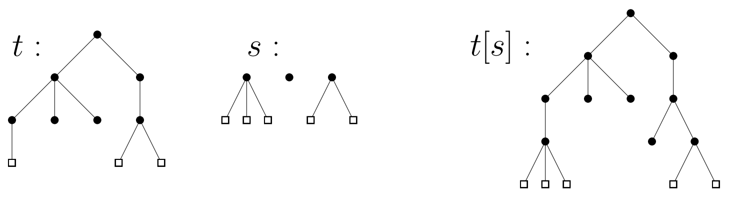


Figure 3: A context t with 3 buds (white squares), a compatible segment s, and the resulting tree $t[s]$. We have $t = stump_{t[s]}(3)$ and $s = segment_{t[s]}(3)$.

3. Automatic Grammar Generation

In this section, we compute an unambiguous algebraic grammar from which any tree over a set S can be derived. More precisely, given a pattern p and a set S, we compute the grammar $G_{S,p} =< N, U, A, S, R >$ where N is the set of non-terminals which are labelled by distinct contexts, $U \subset N$ is a subset of non-terminals whose label contains an occurrence of p, $A \in N$ is the axiom, and R is the set of derivation rules. A rule is composed of four elements (as shown in Figure 4):

1. a non-terminal n,
2. a segment s whose forest contains exactly $|buds(n)|$ trees,
3. a partition of $buds(n[s])$,
4. a tuple of non-terminals.

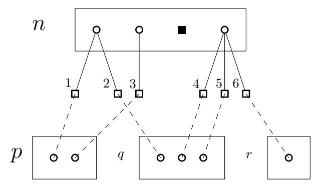


Figure 4: Example of a rule. n, p, q and r are non-terminals of the grammar. The rule can be written

$(n, ($ $,$ $, \blacksquare,$ $), \{\{1, 3\}, \{2, 4, 5\}, \{6\}\}, (p, q, r))$

Figure 6 contains an example of an automatically generated grammar. Figure 7 shows how a tree is derivated in this grammar. Note that this grammar is not a regular tree grammar [7], in which a non-terminal can be labeled by a context with at most 1 bud. In fact, this model generalizes regular tree grammar, and allows to deal with forest, though model will not be described in this paper[2]. To use an analogy, if the studied combinatorial objects were words over an alphabet Σ and the pattern a factor u, we would be constructing the automaton recognizing $\Sigma^* u$ [8]. Each state of this automaton is labeled by a prefix of u. To count the number of occurrences of u in a word w, we would read the prefixes of w in the automaton to check whether they contain u as a suffix. For each prefix v of w, the current state would be labelled by the longest suffix of v which is also a prefix of u.

The idea of our algorithm is very similar. Each stump of the pattern will correspond to a non-terminal in the grammar. Though in some "bad cases", some additional non-terminal will be added. The main difference is that a stump of a tree might contain several independent suffixes.

Definition 6 (Overlapping)**.** *Given a context t and a pattern p, an overlapping is a suffix of t which contains (as a pattern) a* **stump of p of the same height***. These overlappings can be seen as candidates of pattern's occurrences. We note $Over_P[t]$ the set of overlappings in a context t for a pattern P.*

Figure 5 shows an example of the overlappings of a pattern p in a context t. Figure 9 shows the stumps of a pattern with their overlappings.

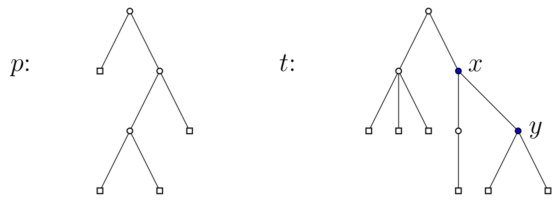


Figure 5: In this figure, p is a pattern and t a context (buds are not relevant here). The two nodes in blue, x and y, are roots of overlappings. x is the root of a tree of height 2 which contains $stump_p(2)$. x is the root of a tree of height 1 which contains $stump_p(1)$.

If one knows the list of overlappings, one can compute which of those trees contains p. Doing this for each stump of a tree t allows to compute the occurrences of p as a pattern. Deriving a stump in the grammar is equivalent to compute this list of suffixes: each suffix is associated to a non-terminal. For readability, a non-terminal will also refer to its associated suffix.

In order to build the grammar, we are going to partition the buds of the tree t according to the maximal (by inclusion) trees s that cover them and which contains $stump_p(height[s])$. The main difficulty of this partition lies in the fact that buds might belong to two subtrees, such that neither of their buds set includes the other.

Figure 6 shows an example of the grammar obtained with our algorithm.

[2]The detailed description of these grammar, which we called Lukasiewicz grammar, is a work in progress.

```
Algorithm 1: Grammar Construction
   Data: A pattern $M$ on a set $S$
   Result: A grammar
           $G =< N, U, A, S, R >$
 1 Compute each segment of $M$;
 2 Compute each stump of $M$ and their
   overlappings;
 3 for each stump st of M do
 4     Add the context st in the list of
       non-terminals $N$;
 5 end
 6 $A = stump_M(0)$;
 7 for each non-terminal n do
 8     $b \leftarrow |buds(n)|$;
 9     for all b-tuple $s \in S^b$ do
10         segment $\leftarrow$ the forest $s$ whose
           leaves are buds;
11         Compute the rule associated to
           $(n, segment)$;
12     end
13     if $height(n) = height(M)$ then
14         Add $n$ in $U$;
15     end
16 end
```

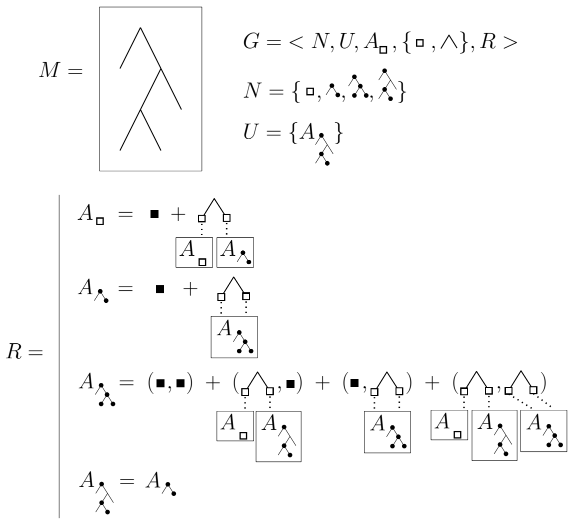


Figure 6: On the left, the algorithm to automatically construct a grammar. On the right, a pattern and the resulting grammar obtained via Algorithm 1. Overlappings roots are distinguished with a small circle. Buds are drawn as white squares and simple leaves (terminal symbols) as black squares. In practice, the rules of a non-terminal t are stored in a tabular of $|buds(t)|$ dimensions.

In order to enhance the readability, the method will be presented the following way:

- Algorithm 1 only presents a backbone of the method. Precalculus are mentioned and we do not explain how a grammar's rule is computed.
- Section 3.1 and 3.2 detail the precalculus. The pattern's segments are precomputed and will be useful to compute the grammar's rules. The pattern's stumps are necessarily non-terminals of the grammar. Because of that, we distinguish them from other non-terminals which can be added in the grammar when computing a rule.
- Section 3.3 contains another algorithm to compute a grammar's rule.

3.1. Compute each segment of the pattern

This precalculus will be useful in order to compute the rules between non-terminals (see Figure 8).

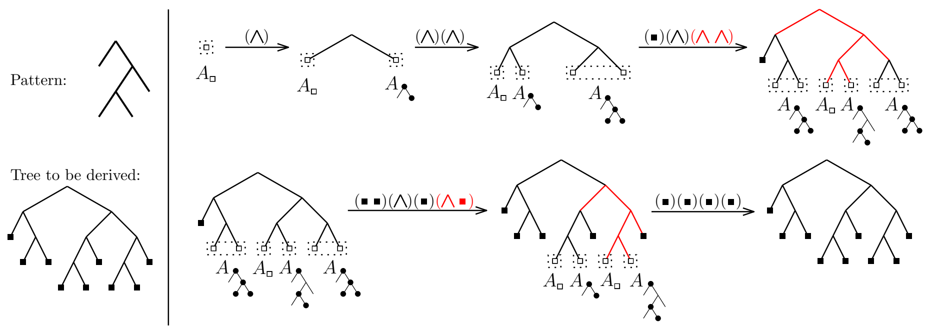


Figure 7: A tree derived by the grammar given in Fig.6, where rules spanning a pattern occurrence are drawn in red.

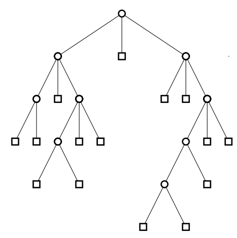
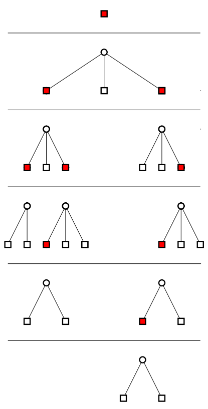



Figure 8: A pattern (on the left) and its segments (on the right). Since each segment is a context, the buds are drawn in red. In order to calculate the segments of a pattern M, one should perform a breadth-first search on it. For each i in $\{0, \ldots height(M) - 1\}$, the buds of $segment_M(i)$ are the internal nodes of height $i + 1$. Such a computation can be done in linear time and space in the size of M.

3.2. Compute the pattern's stumps and their overlappings

The purpose of this step is to compute some non-terminals of the grammar. As we will see, the grammar might contain non-terminals which are more complex than stumps. Those non-terminal will be added when they first appear in a rule (line 10 of Algorithm 1). Though, each stump of the pattern is necessarily a non-terminal, which is why we can compute them from the beginning of the algorithm.

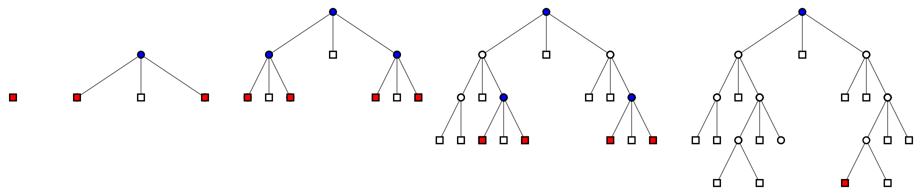


Figure 9: Stumps of pattern from Figure 8 and their overlappings roots (blue). Each overlapping corresponds to another stump. The buds (red) of each stump are obtained by computing the union of the buds of each overlapping.

3.3. Compute the rules of the grammar

A rule in the grammar indicates how, being in a non-terminal n (i.e. a context), and doing a substitution with a given compatible segment s, one does obtain a tuple of non-terminals. The idea is to partition $buds(n[s])$ as represented in Figure 10: each part is a set of buds, maximal by inclusion, contained in a suffix of n which is also a non-terminal.

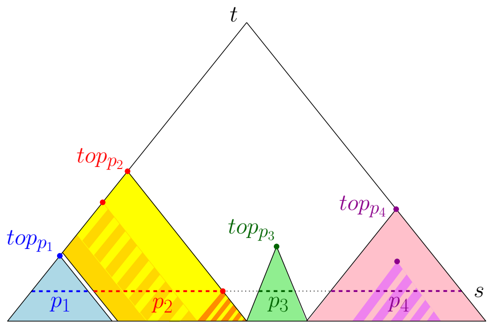


Figure 10: Given a non-terminal labelled by a tree t and a compatible segment s, we partition the buds of $t[s]$. An overlapping is associated to a part if its buds belongs to this part. Each part are covered by a tree enrooted in its highest associated overlapping. Hashed zones represent nodes included in several overlappings.

- Algorithm 2 computes a rule from a non terminal n labelled by a segment s. The idea is to partition the buds of $n[s]$ such that each part P can be associated to a non-terminal n' of the grammar, which appears as a suffix in $n[s]$ and such that $buds(n') = P$ (see Figure 10).
- Section 3.3.1 details how to compute the overlappings of $n[s]$. An overlapping associated to a context t is said to be *continued* by a segment s, if $t[s]$ is already a non-terminal or if t is a stump of height h and $s = segment_p(h)$.
- Section 3.3.2 explain how to partition the set of overlappings according to their buds. The "buds of an overlapping" are the buds of the associated non-terminals.
- Section 3.3.3 shows an example of how a non-terminal which is not a stump can appear in the grammar.

```
Algorithm 2: Compute the rule associated to $(n, s)$
   Data: A Pattern $M$, a grammar $G =< N, U, A, S, R >$, a non-terminal $n$, a
         segment $s$
1  Compute $n[s]$;
2  $Over$ ← Compute overlappings of $n[s]$;
3  $\mathcal{P}$ ← Partition $Over$ according to their buds;
4  $\mathcal{Q} \leftarrow \emptyset$;
5  for all $P_i \in \mathcal{P}$ do
6      $C_i$ ← Compute the highest overlapping in $P_i$;
7      $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\bigcup_{c \in P_i} buds(c)\}$;
8      si $C_i \notin N$ alors
9          Add $C_i$ in $N$;
10     fin
11 end
12 Add $(n, s, \mathcal{Q}, (C_1, \dots, C_{|\mathcal{P}|}))$ in $R$;
```

Let n be a non-terminal whose overlappings have been computed and s a compatible segment. When one substitutes a segment to a non-terminal, one obtains a new context: to compute the rule associated to (n, s) is to decompose this context into distinct non-terminals. To find the non-terminals we need to look at overlappings: in the new tree, there can be overlappings of two kinds:

- old overlappings continued by s: let t be an overlapping in n, of height h. One then has to check if s (associated to $buds(p)$) is equal to $segment[h]$. What occurs at other buds is irrelevant for this overlapping.
- buds of $n[s]$ which are not covered by another overlapping.

There can be no other overlapping since what occurs to the rest of the tree is independent by definition: otherwise, due to the tree structure, the root of n would be included in a larger non-terminal, which is a contradiction.

Once we know where the new overlappings are , we have to compute two partitions: a partition $\mathcal{P}$ of dependent overlappings and a partition $\mathcal{O}$ of $buds(s)$. Each part $O_i \in \mathcal{O}$ is associated to a part $P_i \in \mathcal{P}$ the following way: $O_i = \bigcup_{c \in P_i} buds(c)$. Two overlappings are dependent if one needs to check (at least) one common bud while computing whether they are continued by a given segment.

Dependent overlappings are dealt with a common non-terminal rooted at the highest root of these overlappings. And the context $n[s]$ can be fully decomposed into distinct non-terminals/context C_i covering each part $Q_i \in \mathcal{Q}$. The rule is thus: $(n, s, \mathcal{Q}, (C_1, \dots, C_{|\mathcal{Q}|}))$.

3.3.1. How to quickly compute the overlappings of $n[s]$

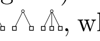
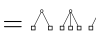
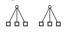
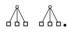
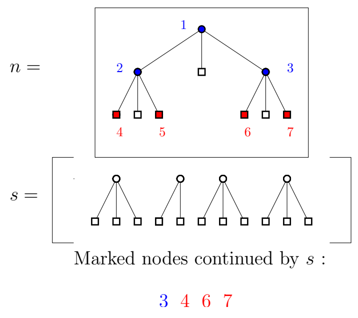
The purpose of this step (Algorithm 3) is to compute the overlappings of a non-terminal n which are still overlappings in $n[s]$. In Figure 11, the buds associated to the overlapping 1 (height 2) are given by the segment of height 2 (i.e. the buds $4, 5, 7$). Hence, $s_1 =$ , whereas $segment_M(3) =$  (See

Figure 8). Thus the overlapping 1 is not continued by s. The buds associated to the overlapping's root 3 (height 1) are given by the segment of height 1 (i.e. the buds $6, 7$). We have $s_3 =$  and $segment_M(2) =$ . Therefore the overlapping 3 is continued by s.



Algorithm 3: Compute a list of overlappings in $n[s]$

```
   Data: A non-terminal $n$ and a segment
         $s \in S^{\mathbb{N}}$
   Result: The list $L$ of $n[s]$ overlappings
1  $L \leftarrow buds(s)$;
2  forall overlapping $m$ of $n$ do
3      $n' \leftarrow$ context associated to $m$;
4      $s_{n'} \leftarrow$ segment $s$ reduced to $buds(n')$;
5      $h \leftarrow height(n')$;
6      if $segment_M(h+1) = s_{n'}$ then
7          Add $m$ in $L$;
8      end
9      return $L$;
10 end
```

Figure 11: On the right, the algorithm to compute the list of overlappings continued by a segment s. On the left, an example for a fixed couple (n, s).

3.3.2. Overlappings Partition

The idea behind this partition is to detect when the branchings in a tree have become *independent*, *i.e.,* when an occurrence of the pattern cannot overlap two parts of the partition. This allows us to control the size of our grammar and, if needed, to parallelize the computation. It also allows to computes the non-terminals which are not stumps of the pattern. Indeed, if the buds of two overlapping intersect without including one another, we need to consider the overlapping of the associated non-terminals as a distinct non-terminal in order to preserve non-ambiguity.

Let m_1, m_2 be two overlapping's roots and c_1, c_2 their associated context. The overlapping's roots m_1 and m_2 are in the same part of the partition if $buds(c_1) \cap buds(c_2) \neq \emptyset$. In Figure 11's example, $Over = (3, 4, 6, 7)$. The buds of the context enrooted in the overlapping's root 3 are 6 and 7. All the other elements in $Over$ are buds. Hence, according to the previously announced rules: $\mathcal{P} = \{\{4\}, \{3, 6, 7\}\}$.

3.3.3. Creation of a non-terminal

A non-terminal in a grammar might not be a stump. Figure 13 shows an example of how such a non-terminal can appear.

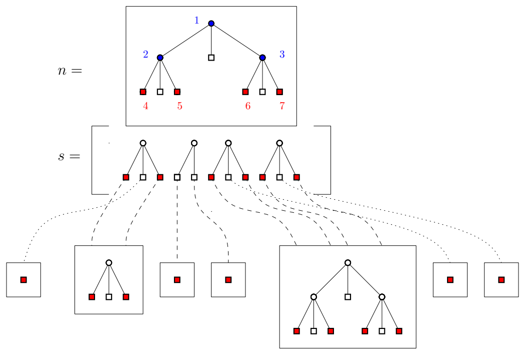


Figure 12: Rule obtained by the algorithm for the couple (n, s) in Figure 11

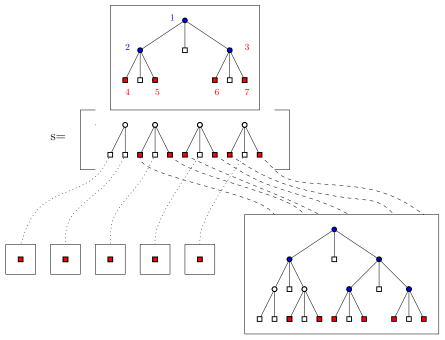


Figure 13: This example differs from Figure 12 because of the segment s. The overlapping's roots continued by s are $1, 3, 5, 6, 7$ and the partition of $Over$ is equal to $\{\{1, 3, 5, 6, 7\}\}$. Since all the buds of n are in a same part P_i, the context C_i contains two overlapping occurrences of $segment_M(3)$ and $segment_M(2)$, which have to be dealt with in a common non-terminal.

3.4. Grammar: properties and complexity

Lemma 1. *The grammar produced by Algorithm 1 has the following properties:*

1. *Each tree defined on a set S can be derived in a unique way.*
2. *The derivation of a tree t containing k occurrences of the pattern p reaches k non-terminals in U.*

Proof. The grammar is deterministic and thus non-ambiguous, meaning there is at most one possible derivation for a tree in our grammar. Any tree can be derived since the grammar is complete and since a leaf is never followed by a non-terminal.

The derivation of a tree can be seen as deriving its stump, one after another. At each step, the set of overlappings (in a stump) is computed and partitioned into subsets closed by inclusion. The biggest overlapping of each subset labels a non-terminal. The non-terminals associated to each stump of t can be computed using the rules in our grammar. Indeed, the overlappings in each stump can be computed using Algorithm 3. From this point, the partition of overlappings and computation of the non-terminals is dealt with Algorithm 2. A non-terminal is in U iff the associated overlapping contains p. According to Algorithm 1, a non-terminal is in U if is labelled by a context of the same height as p, meaning it is either labelled by p itself or a context obtained by Algorithm 2 when computing a rule. This context is necessarily enrooted in an overlapping of the same height as p, ergo it includes p. $\square$

Lemma 2. *The grammar produced by Algorithm 1 has the following size properties:*

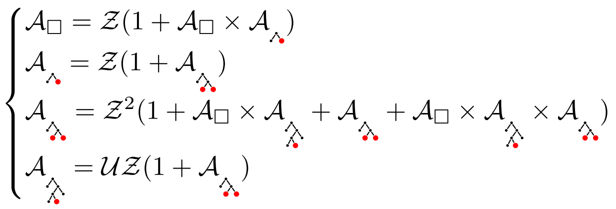
- *the number of non-terminals is logarithmic in the best case scenario and exponential in the worst case scenario, in the size of the pattern.*
- *the number of rules is equal to* $\sum_{t \in N} |S|^{buds(t)}$.

The number of rules is therefore at least exponential in the maximal number of internal nodes of a given height the pattern (i.e. its stumps maximal number of buds).

Proof. The number of non-terminals is lower bounded by the pattern's number of stumps. Since a non-terminal is associated to a tree in which a stump of the pattern might be enrooted, the height of this tree is bounded by the pattern's height. Therefore the number of non-terminal is bounded by the number of trees whose height is less or equal to the pattern's height. The number of rules is given by the number of buds of each non-terminal. $\square$

4. From the Grammar to a Random Sampler

We will briefly sketch how to use our grammar with two classical automatic methods: recursive method [12] and Boltzmann sampling [9, 10]. Both methods are usually described in the *symbolic method* [11] formalism, which is based on specifications. Grammars obtained by Algorithm 1 can easily be transformed into specifications from which we can automatically derive a system over its generating function [11]. Each non-terminal t of the grammar is associated to an equation, which is a product of $\mathcal{Z}^r$ (where r is the size of segments compatible with t) and the union of all the rules R_t. When $t \in U$, we just add a multiplicative monomial $\mathcal{U}$ to the equation. For example, the grammar given in Figure 6 leads to the specification:



Using the recursive method, one can directly use our grammar to randomly generate trees with exactly n nodes and exactly m occurrences of the pattern. To do so, one needs to compute and store the coefficients of the bivariate generating function.

The Boltzmann method requires the grammar to be transformed into a generating function and a manual study of its main singularity. Gladly, this can already be done automatically for monovariate exponential generating function using a tool like Mapple, and the proper packages: **Combstruct** and **NewtonGF** [3][16].

The study of main singularity in a bivariate function still requires a "manual" work. Once this is done, we can obtain another kind of sampler: for any $\varepsilon > 0$, the bivariate Boltzmann sampler [5] draws trees of size $n \pm \varepsilon$ and of number of occurrences approximately m (the precision depends on n). Its average complexity is $\Theta(n)$, because the average distribution of pattern is Gaussian [6] for the free sampler. In both cases, the sampling is always efficient if the size of the pattern is negligible compared to the size of the trees we want to sample.

Remark 1. *Note that the random sampler can be generalized by controlling the average number of occurrences of nodes with a given arity, as it has been done in [5, 1].*

A factorized representation of the generating function :. The main difference between the grammar and the generating function is the quantity of information that needs to be kept. We can obtain a non-linear equation system, in which each line corresponds to a non-terminal in the grammar. A line in this system, for a pattern M, can be written the following way:

$$N = T - A + B,$$

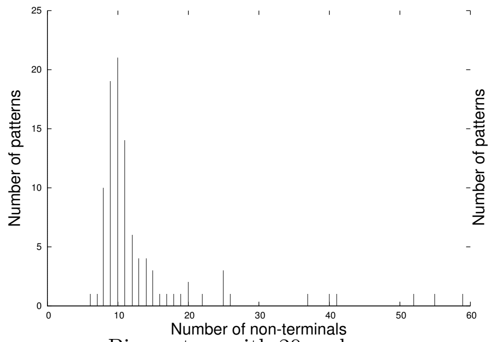
- N is a non-terminal in the grammar.
- T is the tuple of non-terminals $Over[N]$ different from the root, maximal by inclusion.
- let S be the set of segments containing $segment_M(height(N))$,
- B is the tuple of rules obtained by substituting T with S.
- A is the tuple of rules obtained by substituting N with S.

5. Experimental Results

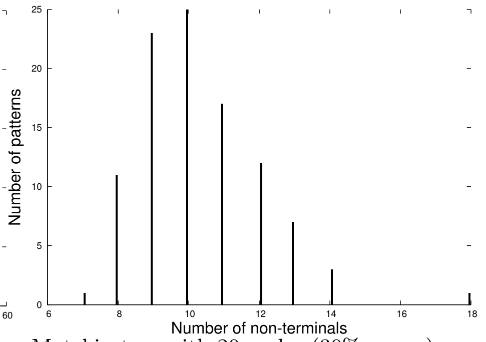
The algorithm has been implemented and tested on random binary trees and binary-ternary trees. For a hundred random binary trees with 20 nodes, there is experimentally always a pattern for which the grammar explodes in such proportion that the program stops. Figure 14 shows an histogram of the number of non terminals when the size doesn't explodes.

The weakness of the algorithm is currently the number of generated rules. Figure 15 shows the proportion/number of patterns amongst a hundred random pattern whose number of rules is included in a given interval.

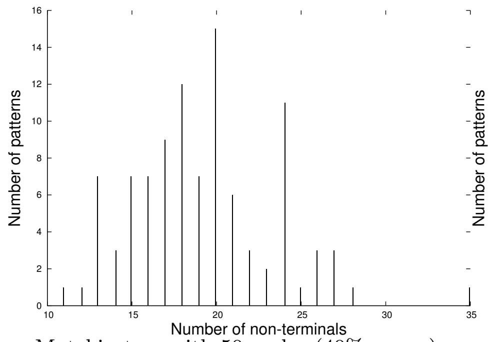
[3]This new package can be downloaded at *http://perso.ens-lyon.fr/bruno.salvy/software/the-newtongf-package/*



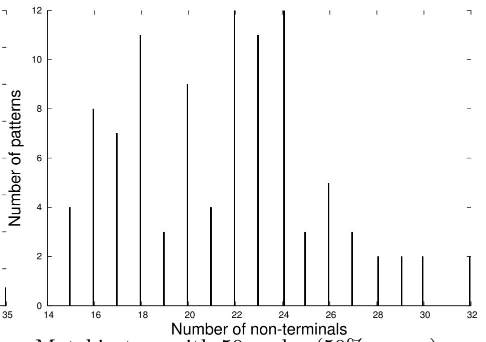
Binary tree with 20 nodes



Motzkin tree with 20 nodes (30% unary)



Motzkin tree with 50 nodes (40% unary)



Motzkin tree with 50 nodes (50% unary)

Figure 14: Number of non-terminals given the algorithm on 100 random pattern of various size and proportion of unary nodes.

Number of rules	≤ 50	≤ 500	≤ 5000	≤ 50000	≤ 400000
Proportion	24	49	16	2	9

Binary tree with 20 nodes.

Number of rules	≤ 200	≤ 500	≤ 3000
Proportion	74	13	13

Motzkin tree with 20 nodes (30% unary).

Number of rules	≤ 500	≤ 1000	≤ 2000	≤ 5000	≤ 50000	≤ 300000
Proportion	25	21	19	20	12	3

Motzkin tree with 50 nodes (40% unary).

Number of rules	≤ 500	≤ 1000	≤ 2000	≤ 5000	≤ 50000	≤ 300000
Proportion	35	28	15	12	9	1

Motzkin tree with 50 nodes (50% unary).

Figure 15: Number of rules given the algorithm on 100 random pattern of various size and proportion of unary nodes.

Those experiments shows that the algorithm is not always efficient. The worst case can be reached even with small sized patterns. Though it also shows that it might also be efficient in a lot of cases. A more detailed analysis of our algorithm's efficiency would require an average study of the overlappings in random trees.

6. From the grammar to a Pattern Matching Algorithm

Using the grammar described in Section 3, we describe an algorithm which counts the occurrences of a patterns p in a tree t with a time complexity $\Theta(|t|)$. This algorithm can be seen as a top-down version of the bottom-up pattern matching algorithm proposed in [14]. Though, since its pre-calculus can be exponential in the size of p, this method will always be useful in a context where $|p|$ is exponentially smaller than $|t|$ and/or to match the same pattern in a very large amount of different tress.

Algorithm 4: Pattern matching algorithm

```
Data: A tree $t$, a grammar $G =< N, U, A, \Sigma, R >$ associated to a tree
      pattern $p$
Result: The number of occurrences of $p$ in $t$
1  a queue of non terminals $QN := push(A)$;
2  a queue of tuple of trees $QT := push(t)$;
3  $result := 0$;
4  while $QN$ is not empty do
5      $N := pop(QN)$;
6      $(t_1, \dots, t_r) := pop(QT)$;
7      if $N \in U$ then
8          $result := result + 1$
9      end
10     $\mathcal{P} := R_N[t_1] \cdots [t_r]$;
11     foreach $P \in \mathcal{P}$ do
12         $QN := push($ the non-terminal associated to $P$ $)$;
13         $QT := push($ tuple formed with the subtrees rooted at $P)$ ;
14     end
15 end
16 return $result$
```

Theorem 1. *The algorithm 16 returns the number of occurrences of p in t. Its time complexity is $\Theta(|t|)$ and its space-complexity is $\mathcal{O}(|t|)$.*

Proof. The algorithm counts the number of times a non-terminals in U is seen, which is equivalent to counting the occurrences of the pattern (Lemma 1). The time and space complexity can be easily seen: each node of t is added in QT exactly once and the number of non-terminals added in QN is bounded by $|t|$. Moreover, each instructions between lines 5 and 13 can be dealt with in constant time using adequate data structures. □

7. Conclusion

The algorithm we introduced can be used in practice on large quantity of patterns. Though, it can still be improved in several ways:

- it can be generalized to a set of pattern,
- one can tremendously decrease the grammar representation using the factorized representation given in Section 4. An algorithm in order to obtain this factorization should be detailed and implemented. This should be done in a future work.
- an average study would be useful to understand when the algorithm is efficient.

References

[1] Cyril Banderier, Olivier Bodini, Yann Ponty, and Hanane Tafat. On the diversity of pattern distributions in rational language. In *Proceedings of the 12th Meeting on Analytic Algorithmics & Combinatorics*, pages 107–116, Kyoto, Japon, 2012. Omnipress.

[2] Frédérique Bassino, Mathilde Bouvel, Adeline Pierrot, Carine Pivoteau, and Dominique Rossin. Combinatorial specification of permutation classes. In *Proceedings of FPSAC 2012 (24th International Conference on Formal Power Series and Algebraic Combinatorics)*, volume AR, pages 781 – 792, Nagoya, Japon, 2012.

[3] Antonio Bernini, Luca Ferrari, Renzo Pinzani, and Julian West. Pattern-avoiding dyck paths. In *Proceedings of FPSAC 2013 (25th International Conference on Formal Power Series and Algebraic Combinatorics)*, Paris, France, 2013.

[4] Olivier Bodini and Alice Jacquot. Boltzmann samplers for colored combinatorial objects. In *Generation Aleatoire de Structure Combinatoire (Gascom'08), Bibbiena, Italy.*, 2008.

[5] Olivier Bodini and Yann Ponty. Multi-dimensional boltzmann sampling of languages. In *DMTCS Proceedings*, pages 49–64, Vienne, Autriche, 2010. 12pp.

[6] Frédéric Chyzak, Michael Drmota, Thomas Klausner, and Gerard Kok. The distribution of patterns in random trees. *Combinatorics, Probability & Computing*, 17(1):21–59, 2008.

[7] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: *http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.*

[8] *Maxime Crochemore and Wojciech Rytter.* Jewels of stringology. *World Scientific, 2002.*

[9] *Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures.* Combinatorics, Probability & Computing*, 13(4-5):577–625, 2004.*

[10] *Philippe Flajolet, Eric Fusy, and Carine Pivoteau. Boltzmann sampling of unlabelled structures. In* Analytic Combinatorics and Algorithms Conference (ANALCO'07)*. SIAM, 2007.*

[11] *Philippe Flajolet and Robert Sedgewick.* Analytic Combinatorics. *Cambridge University Press, 2009.*

[12] *Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus of random generation of labelled combinatorial structures.* Theoret. Comput. Sci.*, 132:1–35, 1994.*

[13] *T. Flouri, B Melichar, and J. Janousek. Subtree matching by deterministic pushdown automata. In* IMCSIT'09*, volume 4, pages 659–666. IEEE Computer Society Press, 2009.*

[14] *Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees.* J. ACM*, 29(1):68–95, 1982.*

[15] *David A. Levin, Yuval Peres, and Elizabeth L. Wilmer.* Markov chains and mixing times*. American Mathematical Society, 2006.*

[16] *Carine Pivoteau, Bruno Salvy, and Michèle Soria. Algorithms for combinatorial structures: Well-founded systems and newton iterations.* Journal of Combinatorial Theory, Series A*, 119(8):1711 – 1773, 2012. Available at http://fr.arxiv.org/abs/1109.2688.*