Enumeration of Unordered Forests

Florian Ingels, Romain Azaïs

Enumeration Of Unordered Forests

Florian Ingels
florian.ingels@inria.fr

Romain Azaïs
romain.azais@inria.fr

Laboratoire Reproduction et Développement des Plantes, Univ Lyon, ENS de Lyon, UCB Lyon 1, CNRS, INRAE, Inria, F-69342, Lyon, France

Abstract

Reverse search is a convenient method for enumerating structured objects, that can be used both to address theoretical issues and to solve data mining problems. This method has already been successfully developed to handle unordered trees. If the literature proposes solutions to enumerate singletons of trees, we study in this article a more general problem, the enumeration of sets of trees – forests. By compressing each forest into a Directed Acyclic Graph (DAG), we develop a reverse search like method to enumerate DAGs compressing forests. Remarkably, we prove that these DAGs are in bijection with the row-Fishburn matrices, a well-studied class of combinatorial objects. In a second step, we derive our forest enumeration to provide algorithms for tackling two related problems : (i) the enumeration of "subforests" of a forest, and (ii) the frequent "subforest" mining problem. All the methods presented in this article enumerate each item uniquely, up to isomorphism.

keywords: Directed Acyclic Graph, Reverse Search, Unordered Trees, Enumeration, Forest

1 Introduction

1.1 Context of the work

Enumeration of trees is a long-term problem, where Cayley was the first to propose a formula for counting unordered trees in the mid-19th century [9, I.5.2]. The exhaustive enumeration of ordered and unordered trees[1] was successfully tackled in the early 00's by Nakano and Uno in [19, 20]. In the unordered case, an extension of the algorithm has been proposed to solve the problem of frequent substructure mining [1]. Moreover, in the field of machine learning, we have recently demonstrated that exhaustive enumeration of the subtrees of a tree makes it possible to design classification algorithms significantly more efficient than their counterpart without such enumeration [5].

Our ambition in this article is to take these two problems of enumeration – trees and subtrees – to a higher order, i.e. to enumerate sets of trees instead of singletons. Specifically, we call an irredundant forest (shortened to forest in the sequel) a set of trees that contains no repetition – in the sense that no tree is a subtree of another (see upcoming Subsection 1.2 for a precise definition). This

[1] A rooted tree is a connected graph without cycles such that there exists a special vertex called the root that has no parent, and the other vertices have exactly one parent. Trees are called ordered or unordered whether the order among siblings is important or not. See Subsection 1.2.

condition of non-repetition is in line with a parsimonious enumeration approach, where the objects considered are all different and enumerated up to isomorphism. Besides, this condition is not restrictive since one can always introduce repetition afterwards. We are therefore interested in the problem of enumerating forests of unordered trees, and then, given a tree or forest, to enumerate all its "subforests" – as forests of subtrees. The latter has already been discussed in the literature, but without consideration on isomorphism [22]. We re-emphasize that we aim to enumerate these various items – forests and subforests – up to isomorphism.

Such an ambition immediately raises a number of obstacles. First of all, the trees are indeed unordered, but so are the sets of trees. For the former the literature has introduced the notion of the canonical form of a tree [20, 1], which is a unique ordered representation of an unordered tree. The enumeration therefore focuses only on these canonical trees. Unfortunately, if it is possible to order a set of vertices, there is no total order on the set of trees, to the best of our knowledge. In addition, the condition of non-repetition filters the set of forests in a non-trivial way, making, a priori, the enumeration problem trickier.

Enumeration problems are recurrent in many fields, notably combinatorial optimization and data mining. They involve the exhaustive listing of a subset of the elements of a search set (possibly all of them), e.g. graphs, trees or vertices of a simplex. Given the possibly high combinatorial nature of these elements, it is essential to adopt clever exploration strategies as opposed to brute-force enumeration, typically to avoid areas of the search set not belonging to the objective subset.

One proven way of proceeding is to provide the search set with an enumeration tree structure; starting from the root, the branches of the tree are explored recursively, eliminating those that do not address the problem. Based on this principle, we can notably mention the well-known "branch and bound" method in combinatorial optimization [18] and the gSpan algorithm for frequent subgraph mining in data mining [26]. Another of these methods is the so-called reverse search technique, which requires that the search set has a partial order structure, and which has solved a large number of enumeration problems since its introduction [2] until very recently [25]. Actually, the algorithms previously introduced in the literature to enumerate trees are based on this technique [19, 20, 1].

In the present paper, we restrict ourselves to reverse search methods, for which the following formalism is adapted from the one that can be found in [21, p. 45-51], and slightly differ from the original definition by Avis and Fukuda [2]. We refer the reader to these two references for further details.

Let $(\mathbb{S}, \subseteq)$ be a partially ordered set, and $g : \mathbb{S} \to \{\top, \bot\}$ be a *property*, satisfying anti-monotonicity

$$\forall s, t \in \mathbb{S} : (s \subseteq t) \wedge g(t) \implies g(s).$$

The *enumeration problem* for the property g is the problem of listing all elements of $E_{\mathbb{S}}(g) = \{s \in \mathbb{S} : g(s) = \top\}$. An enumeration algorithm is an algorithm that returns $E_{\mathbb{S}}(g)$.

The reverse search technique relies on inverting a *reduction rule* $f : \mathbb{S} \setminus \emptyset \to \mathbb{S}$, where f satisfies the two properties of (i) *covering*: $\forall s \in \mathbb{S} \setminus \emptyset, f(s) \subset s$ and (ii) *finiteness*: $\forall s \in \mathbb{S} \setminus \emptyset, \exists k \in \mathbb{N}^*, f^k(s) = \emptyset$. Then, the *expansion rule* is defined as $f^{-1}(t) = \{s \in \mathbb{S} : f(s) = t\}$. This defines an enumeration tree rooted in $\emptyset$, and repeated call to f^{-1} can therefore enumerates all the elements of $\mathbb{S}$.

The reverse search algorithm is shown in Algorithm 1. $E_{\mathbb{S}}(g)$ can be obtained from the call of REVERSESEARCH$((\mathbb{S}, \subseteq), f^{-1}, g, \emptyset)$. As g is anti-monotone, if $g(s) = \bot$, then all elements $s \subseteq t$ also have $g(t) = \bot$, and thefore pruning the enumeration tree in s does not miss any element of $E_{\mathbb{S}}(g)$.

Algorithm 1: REVERSESEARCH

Input: $(\mathbb{S}, \subseteq)$, f^{-1}, g, $s_0 \in \mathbb{S}$ – s.t. $g(s_0) = \top$

```
output s_0
for t ∈ {s ∈ f^{-1}(s_0) | g(s) = ⊤} do
    ReverseSearch((S, ⊆), f^{-1}, g, t)
```

When successfully designed, a reverse search technique should yield polynomial output delay [15, 2], i.e., the time between the output of one element and the next is bounded by a polynomial function in the size of the input.

Remark 1.1. *It would have been possible to define directly the set $\mathbb{S}$ as the set of elements verifying the property g. Separating the two induces that the reduction rule f formally depends only on $\mathbb{S}$, and not on g. This allows, once f is constructed once and for all, to filter $\mathbb{S}$ according to various properties g without additional work. In particular, this is useful in the case where g depends on a tunable parameter – as in the* frequent pattern mining problem *introduced in Section 6.*

1.2 Precise formulation of the problem

A rooted tree T is a connected graph with no cycle such that there exists a unique vertex called the root, which has no parent, and any vertex different from the root has exactly one parent. Rooted trees are said unordered if the order between the sibling vertices of any vertex is not significant. As such, the set of children of a vertex v is considered as a multiset and denoted by $\mathcal{C}(v)$. The leaves $\mathcal{L}(T)$ are all the vertices without children. The height of a vertex v of a tree T can be recursively defined as

$$\mathcal{H}(v) = \begin{cases} 0 & \text{if } v \in \mathcal{L}(T), \\ 1 + \max_{u \in \mathcal{C}(v)} \mathcal{H}(u) & \text{otherwise.} \end{cases} \tag{1}$$

The height $\mathcal{H}(T)$ of the tree T is defined as the height of its root. The outdegree of a vertex $v \in T$ is defined as $\deg(v) = \#\mathcal{C}(v)$[2]; the outdegree of T is then defined as $\deg(T) = \max_{v \in T} \deg(v)$. The depth of a vertex v is the number of edges on the path from v to the root of the tree.

Two trees T_1 and T_2 are isomorphic if there exists a one-to-one correspondance ϕ between the vertices of the trees such that (i) $u \in \mathcal{C}(v)$ in $T_1 \iff \phi(u) \in \mathcal{C}(\phi(v))$ in T_2 and (ii) the roots are mapped together. For any vertex v of T, the subtree $T[v]$ rooted in v is the tree composed of v and all its descendants – denoted by $\mathcal{D}(v)$. $\mathcal{S}(T)$ denotes the set of all distinct subtrees of T, which is the quotient set of $\{T[v] : v \in T\}$ by the tree isomorphism relation. In this article, we consider only unordered rooted trees that will simply be called trees in the sequel. We denote by $\mathcal{T}$ the set of all trees.

As mentioned before, we are interested in this paper in the enumeration of *forests*. The literature acknowledges two definitions for a forest [6, p. 172]: (i) an undirected graph in which any two vertices are connected by at most one path or (ii) a disjoint union of trees. We adopt a variation of the latter one, that forbids repetitions inside the forest.

[2]The notation # is used in this paper to denote both (i) the cardinality #S of any set S, and (ii) the number of vertices #G of any graph G.

Definition 1.2. *A set* $\{T_1, \ldots, T_n\}$ *of trees is an irredundant forest if and only if*

$$\forall i \neq j, T_i \notin \mathcal{S}(T_j). \tag{2}$$

We denote by $\mathcal{F}$ the set of all irredundant forests – shortened to forests in the sequel of the paper. Our goal is to provide a reverse search method that outputs $\mathcal{F}$. As already stated, this goal raises two major difficulties: firstly, the twofold unordered nature of forests (the set of trees and the trees themselves), and secondly, the non-trivial condition of non-repetition. While the latter problem is intrinsic, the main idea of this paper to address the former is to resort to the reduction of a forest into a Directed Acyclic Graph (DAG).

DAG reduction is a method meant to eliminate internal repetitions in the structure of trees and forests of trees. Beginning with [23], DAG representations of trees are also much used in computer graphics where the process of condensing a tree into a graph is called object instancing [12]. A precise definition of DAG reduction of trees, together with algorithms to compute it, are provided in [10], whereas one technique to extend those algorithms to forests is presented in [5, Section 3.2]. DAG reduction can be interpreted as the construction of the quotient graph of a forest by the tree isomorphism relation. However, in this paper, we provide the general idea of DAG reduction as a vertex coloring procedure.

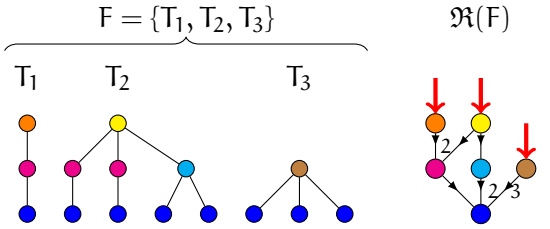


Figure 1: A forest F (left) and its DAG reduction (right). Roots of isomorphic subtrees are identically colored, as well as the corresponding vertex of the DAG. The sources of the DAG (indicated with red arrows) correspond exactly to the roots of the trees in F. For the sake of clarity, arcs of multiplicity greater than one are drawn only once and their multiplicity is written next to the arc.

Consider a forest $F = \{T_1, \ldots, T_n\}$ to reduce. Each vertex of each tree is given a color such that if two distinct vertices u, v belonging respectively to T_i, T_j (not necessarily distinct) have the same color, then $T_i[u]$ and $T_j[v]$ are isomorphic. Reciprocally, if two subtrees are isomorphic, their roots have to be identically colored. Let us denote $c(\cdot)$ the function that associates a color to any vertex. Then, we build a directed graph $D = (V, A)$ with as many vertices as colors used, i.e. $\#V = \#\mathrm{Im}\,(c)$. For any two vertices u, v in the forest, if $u \in \mathcal{C}(v)$, then we create an arc $c(v) \to c(u)$ in D. Note that this definition implies that multiples arcs are possible in D, as if there exist $u, u' \in \mathcal{C}(v)$, for $v \in T$, such that $T[u]$ and $T[u']$ are isomorphic, then the arcs $c(v) \to c(u)$ and $c(v) \to c(u')$ are identical. The graph D is a DAG [10, Proposition 1], i.e. a connected directed (multi)graph without cycles. We refer to Figure 1 for an example of DAG reduction.

In this paper, $\mathfrak{R}(F)$ denotes the DAG reduction of F. It is crucial to notice that the function $\mathfrak{R}$ is a one-to-one correspondence [10, Proposition 4], which means that DAG reduction is a lossless compression algorithm. Since F fulfills condition (2), no tree of F is a subtree of another. If this were the case, say $T_i \in \mathcal{S}(T_j)$, then $\mathfrak{R}(T_i)$ would be a subDAG of $\mathfrak{R}(T_j)$, and therefore the numbers of roots in $\mathfrak{R}(F)$ would be strictly less than #F. Since such a situation can not occur, there are exactly as many roots in $\mathfrak{R}(F)$ as there are elements in F: no information is lost. In other words, F can be reconstructed from $\mathfrak{R}(F)$ and $\mathfrak{R}^{-1}$ stands for the inverse function.

The DAG structure inherits of some properties of trees. For a vertex v in a DAG D, we will denote by $\mathcal{C}(v)$ the set of children of v. $\mathcal{H}(v)$ and $\deg(v)$ are inherited as well. Similarly to trees, we denote

by $\mathrm{D}[v]$ the subDAG rooted in v composed of v and all its descendants $\mathcal{D}(v)$. Note that since $\mathrm{D}[v]$ has a unique root v, it compresses a forest made of a single tree. For the simplicity of notation, we use $\mathfrak{R}^{-1}(\mathrm{D}[v])$ to designate the tree compressed by $\mathrm{D}[v]$ – instead of the singleton.

In the sequel, DAGs compressing forests are called *FDAGs*, to distinguish them from general directed acyclic graphs.

Since DAG compression is lossless, and since a forest can be reconstructed from its DAG reduction, it should be clear that enumerating all forests is equivalent to enumerating all FDAGs. Yet, the latter approach has the merit of transforming set of trees into unique objects, which makes it possible, if able to design a canonical representation – like the trees in [20, 1], to get rid of the twofold unordered nature of forests, as claimed earlier. Indeed, any ordering of the vertices of the DAG induces an order on the roots of the DAG, and therefore on the elements of the forest, as well on the vertices of the trees themselves.

1.3 Aim of the paper

To the best of our knowledge, the enumeration of DAGs has never been considered in the literature. The aim of this article is twofold, i.e (i) to open the way by presenting a reverse search algorithm enumerating FDAGs, in Section 2, and (ii) to derive from it an algorithm for enumerating substructures in Section 5. The frequent pattern mining problem is a classical data mining problem – see [11] for a survey on that question – and we provide in Section 6 a slight variation of the algorithm of Section 5 to tackle this issue. In addition, Section 3 analyses the growth of the enumeration tree defined in Section 2, while Section 4 proposes two variations of it. In more detail, our outline is as follows:

- The first step is to introduce a canonical form for FDAG. For trees [20, Section 3], this consisted in associating an integer (its depth) to each vertex, and maximizing the sequence by choosing an appropriate ordering over the vertices. The notion of depth does not apply to FDAGs, which forces us to find another strategy. DAGs are characterized by the existence of a topological ordering [17], and we introduce in Subsection 2.2 a topological ordering that is unique if and only if a DAG compresses a forest. This canonical ordering is defined so that the sequence of children of the vertices is strictly increasing, where the multisets of children are ordered by the lexicographical order. In fact, these ordered multisets of children are considered as formal words, which brings us to a detour through the theory of formal languages in Subsection 2.1 to introduce useful results for the rest of the article. Compared to trees, we have here a first gain in complexity insofar as we maximize a sequence of words instead of a sequence of integers.

- The expansion rule used for trees [20, Section 4] is to add a new vertex in the tree as a child of some other vertex, so that the depth-sequence remains maximal. Consequently, a single arc is also added. On the other hand, for a FDAG, we want to be able to add either vertices or arcs independently. In Subsection 2.3, we define three expansion rules, reflecting the full spectrum of possible operations, so that the DAG obtained afterward is still a FDAG. Specifically, the branching rule allows to add an arc, where the elongation and widening rules add vertices at different height. We show in Proposition 2.10 that the rules preserve the canonicalness and in Proposition 2.11 that they are "bijective": any FDAG can be reached by applying the

expansion rules to a unique FDAG. In Subsection 2.5 we derive from them an enumeration tree covering the set of FDAGs.

- Notably, a bijection between FDAGs and row-Fishburn matrices, a class of combinatorial objects much exploited in the literature [13, Section 2], is shown in Theorem 3.1 – which proof lies in Appendix A. The asymptotic behavior of these matrices being well known [14, 7], this allows us to derive from it the behavior of the enumeration tree. In return, since our bijection is constructive, the enumeration tree can be used to enumerate row-Fishburn matrices – and all the objects they are in bijection with – via the reverse search method. Remarkably, this bijection operates between two objects that, at first sight, have little in common.

- For an enumeration algorithm to have any practical interest, it is necessary that the associated enumeration tree has a "reasonable" growth – with regard to the size of the explored space. This is the case for our algorithm since we prove, in Subsection 3.2, that a FDAG with n vertices has a number of successors in the enumeration tree in the order of $\Theta(n)$ – and that those successors can be computed in quasi-quadratic time. We also show, in Theorem 3.6, that our algorithm runs with polynomial delay [15].

- Subsection 4.1 introduces a way of enumerating forests in their classical definition, i.e., with redundancy, where some trees may be equal to or subtrees of others. The proposed method takes a redundancy-free forest, as enumerated by our algorithm, and adds repetition in an extra enumeration step. Finally, Subsection 4.2 concludes on enumeration by proposing sets of constraints that make the enumeration tree finite. Indeed, since the rules only allow to increase the height, degree or number of vertices, it is sufficient to set maximum values for some of these parameters to achieve this goal; however the combination of parameters has to be wisely chosen, as we show it.

- Since the structures we enumerate are forests, it is natural that the substructures we are interested in are "subforests". A precise definition of the latter is given in Section 5, i.e. forests of subtrees, and are referred to as subFDAGs. An algorithm to enumerate all subFDAGs appearing in a FDAG is also provided. The frequent subFDAG mining problem is finally addressed in Section 6.

Concluding remarks concerning the implementation of our results in the Python library `treex` [3] are briefly mentioned at the end of the article. In Appendix B, the interested reader will find an index of frequent notations used throughout the paper.

2 Exhaustive enumeration of FDAGs

In this section, we introduce our main result, that is, a reverse search algorithm for the enumeration of FDAGs. As we will consider the multisets of children of vertices as formal words built on the alphabet formed by the set of vertices, we introduce in Subsection 2.1 some definitions and results on formal languages that will be useful for the sequel. We characterize unambiguously in Subsection 2.2 our objects of study, through the lens of topological orderings, defining a canonical topological ordering for DAGs, that is unique if and only if a DAG compresses a forest of unordered trees, i.e. it is a FDAG – see Theorem 2.4. We then define three expansion rules that are meant to extend the structures of FDAGs in Subsection 2.3, and we study their properties in Subsection 2.4.

In Subsection 2.5, we show with Theorem 2.12 that these expansion rules define an enumeration tree on the set of FDAGs.

2.1 Preliminary: a detour through formal languages

We present in this subsection some definitions and results on formal languages that will be useful for the sequel of Section 2.

Let $\mathcal{A}$ be a totally ordered finite set, called alphabet, whose elements are called letters. A word is a finite sequence of letters of $\mathcal{A}$. The length of a word w is equal to its number of letters and is denoted by $\#w$. There is a unique word with no letter called the empty word and denoted by ϵ. The set of all words is denoted by $\mathcal{A}^*$. Words can be concatenated to create a new word whose length is the sum of the lengths of the original words; ϵ is the neutral element of this concatenation operation.

The lexicographical order over $\mathcal{A}^*$, denoted by $<_{\text{lex.}}$ is defined as follows. Let $w_1 = a_0 \cdots a_p$ and $w_2 = b_0 \cdots b_q$ be two words, with $a_i, b_j \in \mathcal{A}$. If $\#w_1 = \#w_2$, then $w_1 <_{\text{lex.}} w_2$ if and only if $\exists k \in [\![0, p]\!], a_i = b_i \ \forall i < k$ and $a_k < b_k$. Otherwise, let $m = \min(p, q)$; $w_1 <_{\text{lex.}} w_2$ if and only if either (i) $a_0 \cdots a_m <_{\text{lex.}} b_0 \cdots b_m$ or (ii) $a_0 \cdots a_m =_{\text{lex.}} b_0 \cdots b_m$ and $m < q$ – that is, $p < q$. Note that, by convention, $\epsilon <_{\text{lex.}} w$ for any word w.

Let $w \in \mathcal{A}^*$. We define the suffix-cut operator $\mathrm{SC}(w)$, which removes the last letter of w:

$$\mathrm{SC}(w) = \begin{cases} w' & \text{if } w = w'a \text{ with } a \in \mathcal{A} \text{ and } w' \in \mathcal{A}^*, \\ \epsilon & \text{otherwise.} \end{cases} \tag{3}$$

A *language* is a set of words satisfying some construction rules. We introduce hereafter two languages that will be useful in the sequel of the paper.

Definition 2.1. *The language of decreasing words is defined as*

$$\Lambda = \left\{ w = a_0 \cdots a_m \in \mathcal{A}^* : a_i \geq_{lex.} a_{i+1} \ \forall i \in [\![0, m-1]\!] \right\}.$$

Definition 2.2. *Let $\overline{w} \in \Lambda$. The language of decreasing words bounded by $\overline{w}$ is defined as*

$$\Lambda^{\overline{w}} = \{ w \in \Lambda : w >_{lex.} \overline{w} \}.$$

Any word $w \in \Lambda^{\overline{w}}$ is said to be minimal if and only if $w \in \Lambda^{\overline{w}}$ but $\mathrm{SC}(w) \notin \Lambda^{\overline{w}}$.

As an example, if $\mathcal{A} = \{0, 1, 2, 3\}$, then $\overline{w} = 211 \in \Lambda$, whereas $121 \notin \Lambda$. In addition, $\Lambda^{\overline{w}}$ contains words such as 31, 22, 21110, etc. 22 is a minimal word of $\Lambda^{\overline{w}}$ as $22 >_{\text{lex.}} 211$ but $\mathrm{SC}(22) = 2 <_{\text{lex.}} 211$.

Our focus is now on the construction of the minimal words of $\Lambda^{\overline{w}}$. Let $\overline{w} = a_0 \cdots a_p$ and $w = b_0 \cdots b_q \in \Lambda^{\overline{w}}$. Taking into account that $w >_{\text{lex.}} \overline{w}$ and that they both are decreasing words, there are only two possibles cases:

(i) w and $\overline{w}$ share a common prefix $a_0 \cdots a_m$. Then $w = a_0 \cdots a_m b_{m+1} \cdots b_q$, and the word $a_0 \cdots a_m b_{m+1}$ is minimal by applying successive suffix-cut operations.

(ii) w and $\overline{w}$ do not share a common prefix. Necessarily $b_0 >_{\text{lex.}} a_0$, and then the word b_0 is minimal by applying several suffix-cut operations.

From the above, we deduce a method for constructing all minimal word of $\Lambda^{\overline{w}}$. First, we partition $\mathcal{A}$ into disjoint – potentially empty – subsets:

$$\begin{aligned}\mathcal{A}^0 &= \{a \in \mathcal{A} : a >_{\text{lex.}} a_0\},\\ \mathcal{A}^i &= \{a \in \mathcal{A} : a_{i-1} \geq_{\text{lex.}} a >_{\text{lex.}} a_i\} \quad 1 \leq i \leq p,\\ \mathcal{A}^{p+1} &= \{a \in \mathcal{A} : a_p \geq_{\text{lex.}} a\}.\end{aligned}$$

It then follows that – empty $\mathcal{A}^i$'s not being considered,

- $\forall b \in \mathcal{A}^0$, the word b is minimal,
- $\forall b \in \mathcal{A}^i$ with $i \in \{1, \ldots, p\}$, the word $a_0 \cdots a_{i-1} b$ is minimal,
- $\forall b \in \mathcal{A}^{p+1}$, the word $\overline{w}b$ is minimal.

As we partitioned $\mathcal{A}$, we have proved the following proposition.

Proposition 2.3. *The number of minimal words of $\Lambda^{\overline{w}}$ is exactly $\#\mathcal{A}$.*

As a follow-up of the example some lines ago, with $\mathcal{A} = \{0, 1, 2, 3\}$ and $\overline{w} = 211$, we apply the proposed method to find the minimal elements of $\Lambda^{\overline{w}}$. We partition $\mathcal{A}$ into: $\mathcal{A}^0 = \{3\}$, $\mathcal{A}^1 = \{2\}$, $\mathcal{A}^2 = \emptyset$, $\mathcal{A}^3 = \{0, 1\}$. The four minimal words are therefore 3, 22, 2111 and 2110.

Although the previous result is completely general, if we require that $\mathcal{A} = \{0, \ldots, n\}$, then the partition method described above can be rewritten into Algorithm 2. While this is not included in the pseudocode provided, note that the algorithm should return an empty list if $a_0 > n$, as in this case there would be no minimal word to look for.

Algorithm 2: MINIMALWORDS

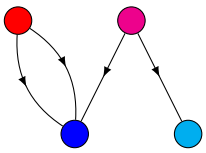
Input: $\overline{w} = a_0 \cdots a_p$, $\mathcal{A} = \{0, \ldots, n\}$
Output: All minimal words of $\Lambda^{\overline{w}}$

```
Set L to the empty list
if a_0 < n then
    for i ∈ {a_0 + 1, ..., n} do
        Add the word i to L
for k ∈ {1, ..., p} do
    if a_k < a_{k-1} then
        for i ∈ {a_k + 1, ..., a_{k-1}} do
            Add the word a_0 ··· a_{k-1} i to L
for i ∈ {0, ..., a_p} do
    Add the word a_0 ··· a_p i to L
return L
```

2.2 Canonical FDAGs

FDAGs are unordered objects, like the trees they compress, and therefore their enumeration requires to reflect this nature. In practice, finding a systematic way to order them makes it possible to design a simpler reduction rule, as done for trees [20], ignoring the combinatorics of permutations. The purpose of this subsection is to provide a unique way to order FDAGs. We show that such an order exists in Theorem 2.4, unambiguously characterizing FDAGs. The approach chosen is based on the notion of topological order.

Topological ordering Let D be a directed graph, where multiple arcs are allowed. A topological ordering on D is an ordering of the vertices of D such that for every arc uv from vertex u to vertex v, u comes after v in the ordering. Formally, $\psi : D \to [\![0, \#D - 1]\!]$ is a topological ordering if and only if ψ is bijective and $\psi(u) > \psi(v)$ for all $u, v \in D$ such that there exists at least one arc uv in D. A well known result establishes that D is a DAG if and only if it admits a topological ordering [17]. Nonetheless, when a topological ordering exists, it is in general not unique – see Figure 2. A reverse search enumeration of topological orderings of a given DAG can actually be found in [2, Section 3.5].



○	🔴	🟣	🔵	🔵
$\psi_1(○)$	3	2	1	0
$\psi_2(○)$	3	2	0	1
$\psi_3(○)$	2	3	0	1
$\psi_4(○)$	2	3	1	0
$\psi_5(○)$	1	3	0	2

Figure 2: The DAG on the left admits five topological orderings, which are shown in the table.

Constrained topological ordering We aim to reduce the number of possible topological orderings of a DAG by constraining them. Let D be a DAG and ψ a topological ordering. Taking advantage of the vertical hierarchy of DAG, our first constraint is

$$\forall (u, v) \in D^2, \quad \mathcal{H}(u) > \mathcal{H}(v) \implies \psi(u) > \psi(v). \tag{4}$$

Applying (4) to the topological orderings presented in Figure 2, ψ_5 must be removed, as $\psi_5(🔵) > \psi_5(🔴)$ and $\mathcal{H}(🔴) > \mathcal{H}(🔵)$.

For any vertex v, and any $u \in \mathcal{C}(v)$, by definition, $\mathcal{H}(v) > \mathcal{H}(u)$. Therefore, there can be no arcs between vertices at same height. Any arbitrary order on them leads to a different topological ordering. The next constraint we propose relies on the lexicographical order:

$$\forall (u, v) \in D^2, \quad \mathcal{H}(u) = \mathcal{H}(v) \text{ and } \mathcal{C}_\psi(u) >_{\text{lex.}} \mathcal{C}_\psi(v) \implies \psi(u) > \psi(v), \tag{5}$$

where $\mathcal{C}_\psi(v)$ is the list $[\psi(v_i) : v_i \in \mathcal{C}(v)]$ sorted by decreasing order w.r.t. the lexicographical order. In other words, $\mathcal{C}_\psi(v)$ is a decreasing word – see Definition 2.1 – on the alphabet $\mathcal{A} = [\![0, \#D - 1]\!]$. Table 1 illustrates the behavior of (5) on the followed example of Figure 2.

○	🔴	🟣	(5)
$\mathcal{C}_{\psi_1}(○)$	11	10	✓
$\mathcal{C}_{\psi_2}(○)$	00	10	✗
$\mathcal{C}_{\psi_3}(○)$	00	10	✓
$\mathcal{C}_{\psi_4}(○)$	11	10	✗

Table 1: Application of (5) to the remaining topological orderings of Figure 2 that satisfy (4). As $\mathcal{C}_\psi(🔵) = \mathcal{C}_\psi(🔵)$, we only need to consider vertices 🔴 and 🟣. As $\psi_i(🔴) > \psi_i(🟣) \iff i \in \{1, 2\}$, the only orderings that are kept are ψ_1 and ψ_3.

The combination of those two constraints imposes uniqueness in all cases except when there exists $(u, v) \in D^2$ such that $\mathcal{C}_\psi(u) = \mathcal{C}_\psi(v)$ and $u \neq v$. It should be clear that if we impose the upcoming condition (6), such a pathological case can not occur.

$$\forall (u, v) \in D^2, \quad u \neq v \implies \mathcal{C}(u) \neq \mathcal{C}(v) \tag{6}$$

Upcoming Theorem 2.4 establishes that a DAG compresses a forest if and only if the topological order constrained by (4) and (5) is unique. In other words, an unambiguous characterization of FDAGs is exhibited.

Theorem 2.4. *The following statements are equivalent:*

(i) D *fulfills (6),*

(ii) there exists a unique topological ordering ψ *of* D *that satisfies both constraints (4) and (5),*

(iii) there exists a unique forest $\mathsf{F} \in \mathcal{F}$ *– cf. (2) – such that* $\mathsf{D} = \mathfrak{R}(\mathsf{F})$,

where $\mathfrak{R}$ *is the DAG reduction operation defined in Subsection 1.2.*

Proof. $(\mathrm{i}) \iff (\mathrm{ii})$ follows from the above discussion. $(\mathrm{iii}) \implies (\mathrm{i})$ follows from the definition of $\mathfrak{R}$. Indeed, if there was two distinct vertices $(u, v) \in \mathsf{D}^2$ with the same multiset of children, they would have been compressed as a unique vertex in the reduction. We now prove that $(\mathrm{i}) \implies (\mathrm{iii})$.

In the first place, if D fulfills (6), then D must admit a unique leaf, denoted by $\mathcal{L}(\mathsf{D})$. Indeed, if there were two leaves l_1 and l_2, we would have $\mathcal{H}(l_1) = \mathcal{H}(l_2) = 0$ but also $\mathcal{C}(l_1) = \mathcal{C}(l_2) = \emptyset$, which would violate (6). Let $r_1, \ldots, r_k$ be the vertices in D that have no parent. We define $\mathsf{D}_1, \ldots, \mathsf{D}_k$ as the subDAG rooted respectively in $r_1, \ldots, r_k$. Then, we define $\mathsf{T}_i = \mathfrak{R}^{-1}(\mathsf{D}_i)$ and $\mathsf{F} = \{\mathsf{T}_1, \ldots, \mathsf{T}_k\}$. The T_i's are well defined as all vertices in D (consequently in D_i) have a different multiset of children, and therefore compress distinct subtrees – i.e. F fulfills (2), therefore $\mathsf{F} \in \mathcal{F}$. Moreover, $\mathsf{D} = \mathfrak{R}(\mathsf{F})$.

In the sequel of the article, we shall only consider FDAGs. Consequently, from Proposition 2.4, they admit a unique topological ordering ψ satisfying both constraints (4) and (5), called *canonical ordering*. Thus, for any FDAG D, the associated canonical ordering ψ will be implicitly defined. The vertices will be numbered accordingly to their ordering, i.e $\mathsf{D} = (v_0, \ldots, v_n)$ with $\psi(v_i) = i$. Finally, as a consequence of constraints (4) and (5), note that D can be partitioned in subsets of vertices with same height, each of them containing only consecutive numbered vertices. Figure 3 provides an example of a FDAG and its canonical ordering.

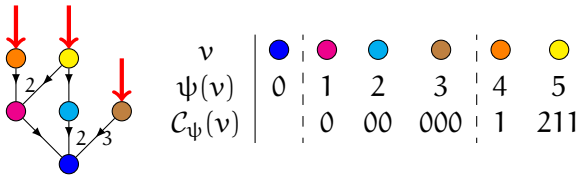


Figure 3: A FDAG D (left) and its canonical ordering ψ (right). Vertices that are at the same height are enclosed in the table between the dashed lines. Red arrows indicate the roots of the trees of the forest that is compressed by D.

2.3 Expansion rules

Reverse search techniques implies finding reduction rules, and then inverse them. Equally, we will define instead three expansion rules, of which inverse will be reduction rules. An expansion rule takes a FDAG and create a new DAG, that is "expanded" in the sense of having either more vertices or more arcs. Our rules are analysed at the end of the subsection, where notably we prove in Proposition 2.10 that expansion rules preserve the canonicalness. Moreover, we show in Proposition 2.11 that they are "bijective": any FDAG is in the image of a unique FDAG through the expansion rules. We begin with a preliminary definition.

Definition 2.5. *Let* D *be a FDAG, with* $D = (v_0, \dots, v_n)$*. We define the two following alphabets*

$$\mathcal{A}_= = \{\psi(v) : v \in D, \mathcal{H}(v) = \mathcal{H}(v_n)\} = \{p+1, \dots, n\},$$
$$\mathcal{A}_< = \{\psi(v) : v \in D, \mathcal{H}(v) < \mathcal{H}(v_n)\} = \{0, \dots, p\},$$

where $p \in [\![0, n-1]\!]$ *and* $\psi(\cdot)$ *is the canonical ordering of* D*.*

In other words, $\mathcal{A}_=$ contains the indices of all vertices that have the same height as the vertex with the highest index according to ψ, and $\mathcal{A}_<$ the indices of all vertices that have an inferior height. The FDAG presented in Figure 3 will serve as a guideline example all along this subsection. Here, we have $\mathcal{A}_= = \{4, 5\}$ and $\mathcal{A}_< = \{0, 1, 2, 3\}$.

The three expansion rules are now introduced. Let $D = (v_0, \dots, v_n)$. Each of these rules is associated with an explicit symbol, which may be used, when necessary, to designate the rule afterward. It is worth noting that all of these rules will operate according to the vertex of highest index, v_n.

Branching rule (↗↖) This rule adds an arc between v_n and a vertex below. The end vertex of the new arc is chosen such that $\mathcal{C}_\psi(v_n)$ remains a decreasing word. In Figure 4, (↗↖) is applied on our guideline example.

Definition 2.6. (↗↖) *Let* $\mathcal{C}_\psi(v_n) = a_0 \cdots a_m$*. Choose* $a_{m+1} \in \mathcal{A}_<$ *such that* $a_m \geq_{lex.} a_{m+1}$ *and add an arc between* $\psi^{-1}(a_{m+1})$ *and* v_n*.*

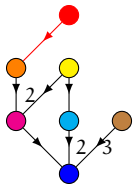
(a) (b)

Figure 4: Branching rule applied to the FDAG of Figure 3. As $\mathcal{C}_\psi(v_5) = 211$, the only letters a we can pick from $\mathcal{A}_< = \{0, 1, 2, 3\}$, satisfying $a \leq_{\text{lex.}} 1$, are 0 and 1. The only two possibles outcomes of (↗↖) are the words (a) 2111 and (b) 2110.

Elongation rule (↕) This rule adds a new vertex v_{n+1} such that $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n)+1$. Consequently, the alphabets change and become $\mathcal{A}_= = \{n+1\}$ and $\mathcal{A}_< = \{0, \dots, n\}$. Note that after using this rule, it is not possible to ever add a new vertex at height $\mathcal{H}(v_n)$. See Figure 5 for an illustration of this rule on the guideline example.

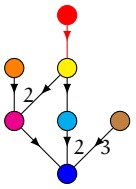
Definition 2.7. (↕) *Add new vertex* v_{n+1} *such that* $\mathcal{C}_\psi(v_{n+1}) = a_0 \in \mathcal{A}_=$*.*

Widening rule (↱) This rule adds a new vertex v_{n+1} at height $\mathcal{H}(v_n)$. The vertex is added with children that respects the canonicalness of the DAG, that is, such that $\mathcal{C}_\psi(v_{n+1}) >_{\text{lex.}} \mathcal{C}_\psi(v_n)$ – as in condition (5). In other terms, denoting $\Lambda_<$ the language of decreasing words on alphabet $\mathcal{A}_<$, and with $\overline{w} = \mathcal{C}_\psi(v_n)$, $\mathcal{C}_\psi(v_{n+1})$ must be chosen in $\Lambda_<^{\overline{w}}$ – see Definition 2.2. However, this set is infinite, so we restrict $\mathcal{C}_\psi(v_{n+1})$ to be chosen among the minimal words of $\Lambda_<^{\overline{w}}$. It follows from the definition of suffix-cut operator $SC(\cdot)$ that, by inverting the said operator, the other words in $\Lambda_<^{\overline{w}}$ can be obtained by performing repeated (↗↖) operations. Finally, this new vertex is added to $\mathcal{A}_=$.



v	...		
$\psi(v)$	...	5	6
$\mathcal{C}_\psi(v)$	...	211	4

(a)



v	...		
$\psi(v)$	...	5	6
$\mathcal{C}_\psi(v)$	...	211	5

(b)

Figure 5: Elongation rule applied to the FDAG of Figure 3. As $\mathcal{A}_= = \{4, 5\}$, there are only two choices leading to (a) $\mathcal{C}_\psi(v_6) = 4$ and (b) $\mathcal{C}_\psi(v_6) = 5$. The alphabets become $\mathcal{A}_< = \{0, \dots, 5\}$ and $\mathcal{A}_= = \{6\}$.

Definition 2.8. (↗) *Add new vertex v_{n+1} such that*

$$\mathcal{C}_\psi(v_{n+1}) \in \left\{ w \in \Lambda_<^{\overline{w}} : w \text{ is a minimal word of } \Lambda_<^{\overline{w}} \right\}$$

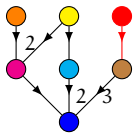
with $\overline{w} = \mathcal{C}_\psi(v_n)$.

From Proposition 2.3 we now that such minimal words exist. We prove in the upcoming lemma that, as claimed, $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n)$.

Lemma 2.9. *Any element of $\Lambda_<^{\overline{w}}$ defines a new vertex v_{n+1} such that $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n)$.*

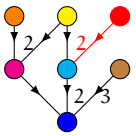
Proof. From the definition of $\mathcal{H}(\cdot)$ – (1), it suffices to prove that v_{n+1} admits at least one child at height $h = \mathcal{H}(v_n) - 1$. Let us denote b_0 and a_0 the first letter of, respectively, $\mathcal{C}_\psi(v_{n+1})$ and $\mathcal{C}_\psi(v_n)$. Denoting $v = \psi^{-1}(b_0)$ and $u = \psi^{-1}(a_0)$, we already know that $\mathcal{H}(u) = h$ – as ψ respects (5) and $\mathcal{C}_\psi(v_n)$ is a decreasing word. Therefore, as by construction $\mathcal{C}_\psi(v_{n+1}) >_{\text{lex.}} \mathcal{C}_\psi(v_n)$, either (i) $b_0 = a_0$ and therefore $v = u$, either (ii) $b_0 >_{\text{lex.}} a_0$. In the latter, as ψ respects (4) and (5), $\mathcal{H}(v) \geq \mathcal{H}(u) = h$. But, as $b_0 \in \mathcal{A}_<$, $\mathcal{H}(v) < \mathcal{H}(v_n) = h + 1$. In both cases, $\mathcal{H}(v) = h$. ∎

Figure 6 illustrates the use of the widening rule on the followed example. It should be noted that the possible outcomes of (↗) are obtained by using Algorithm 2, applied to $w = \mathcal{C}_\psi(v_n)$ and p – with $\mathcal{A}_< = \{0, \dots, p\}$.



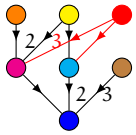
v	...		
$\psi(v)$	...	5	6
$\mathcal{C}_\psi(v)$	...	211	3

(a)



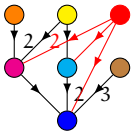
v	...		
$\psi(v)$	...	5	6
$\mathcal{C}_\psi(v)$	...	211	22

(b)



v	...		
$\psi(v)$	...	5	6
$\mathcal{C}_\psi(v)$	...	211	2111

(c)



v	...		
$\psi(v)$	...	5	6
$\mathcal{C}_\psi(v)$	...	211	2110

(d)

Figure 6: We apply (↗) to the FDAG of Figure 3. Here, $\mathcal{A}_< = \{0, 1, 2, 3\}$ and $\overline{w} = 211$. As seen in Subsection 2.1, the minimal words of $\Lambda_<^{\overline{w}}$ are 3, 22, 2111 and 2110. Therefore, there are 4 ways to add a new vertex v_6 via the widening rule, that are such that (a) $\mathcal{C}_\psi(v_6) = 3$, (b) $\mathcal{C}_\psi(v_6) = 22$, (c) $\mathcal{C}_\psi(v_6) = 2111$ or (d) $\mathcal{C}_\psi(v_6) = 2110$. Finally, we update $\mathcal{A}_=$ to be equal to $\{4, 5, 6\}$.

2.4 Analysis of the rules

Since our goal is to enumerate FDAGs, it is required that the expansion rules indeed construct FDAGs. This is achieved by virtue of the following proposition.

Proposition 2.10. *The expansion rules preserve the canonicalness property.*

Proof. Let $\mathrm{D} = (v_0, \dots, v_n)$ be a FDAG. The proposition follows naturally from the definitions:

- (🡅) Let a be the letter added to $w = \mathcal{C}_\psi(v_n)$. As $wa >_{\text{lex.}} w >_{\text{lex.}} \mathcal{C}_\psi(v_{n-1})$, the ordering is unchanged.
- (↕) The new vertex v_{n+1} is such that $\mathcal{H}(v_{n+1}) > \mathcal{H}(v_n)$, so condition (4) is still met.
- (🡇) The new vertex v_{n+1} is chosen so that $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n)$ and $\mathcal{C}_\psi(v_{n+1}) >_{\text{lex.}} \mathcal{C}_\psi(v_n)$, so condition (5) is also still met.

Therefore, any DAG obtained from D is still a FDAG. ∎

Secondly, since our goal is to provide the FDAGs space with an enumeration tree, which will be explored via the expansion rules, it is important that these expansion rules are "bijective" in the following sense: for any FDAG D, there exists a unique FDAG D' such that D is obtained from D' via one of the three rules (🡅), (↕) or (🡇).

Such D' can be constructed via Algorithm 3 as shown in upcoming Proposition 2.11. Conditional expressions applied to D are used to determine which modification should be applied to construct D'. The gray symbol (in the algorithm) next to these modifications indicates which expansion rule allows to retrieve D from D'.

Proposition 2.11. *Algorithm 3 applied to any FDAG constructs the unique antecedent of this FDAG.*

Algorithm 3: ANTECEDENT

Input: $\mathrm{D} = (v_0, \dots, v_n)$; $w = \mathcal{C}_\psi(v_n)$; $w' = \mathcal{C}_\psi(v_{n-1})$

if v_n **is the only vertex of height** $\mathcal{H}(v_n)$ **then**
  if $\#w = 1$ **then**
    (↕) Delete vertex v_n
  else
    (🡅) $w \leftarrow \mathrm{SC}(w)$
else
  if w **is a minimal word of** $\Lambda_{<}^{w'}$ **then**
    (🡇) Delete vertex v_n
  else
    (🡅) $w \leftarrow \mathrm{SC}(w)$

$\mathrm{SC}(\cdot)$ is the suffix-cut operator defined in (3).

Proof. Let $\mathrm{D} = (v_0, \dots, v_n)$ be a FDAG. Let $w = \mathcal{C}_\psi(v_n)$ and $w' = \mathcal{C}_\psi(v_{n-1})$. Two cases can occur: (i) either v_n is the only vertex at height $\mathcal{H}(v_n)$, (ii) or it is not.

(i) It is clear in this case that D can not be obtained from any FDAG via the rule (🡇) – otherwise v_n would not be alone at its height. Concerning (🡅) and (↕), let us look at the number of children of v_n.

  (a) If v_n admits only one child, it must come from an (↕) step, since (🡅) would imply that $\#w \geq 2$. Therefore, in this case, D can be retrieved among the outcomes of rule (↕) applied to $\mathrm{D}' = (v_0, \dots, v_{n-1})$.

(b) Otherwise, when $\#w > 1$, D can not come from an (↕) step, and must therefore come from (🌂). Denoting v'_n the vertex with list of children $SC(w)$ – see (3), D is one of the outcomes of $D' = (v_0, \dots, v_{n-1}, v'_n)$ via (🌂).

(ii) following the same logic as (i), D can not be obtained via (↕). We discrimine between rules (🏂) and (🌂) be comparing w and w'. If w is a minimal word of $\Lambda^{w'}_{<}$, then D can not be obtained from (🌂) – this would break the canonical order. Therefore, in this case, D is an outcome of rule (🏂) applied to $D' = (v_0, \dots, v_{n-1})$. Otherwise, if w is not a minimal word, then it can not be obtained from (🏂), and must come from a (🌂) step, applied to $D' = (v_0, \dots, v_{n-1}, v'_n)$ where $\mathcal{C}_\psi(v'_n) = SC(w)$.

Whatever the case among those evoked, they correspond exactly to the conditional expressions of the Algorithm 3, which therefore constructs the correct antecedent of D, which is unique by virtue of the previous discussion. 🪶

2.5 Enumeration tree

In this subsection, we construct the enumeration tree of FDAGs derived from the expansion rules of Subsection 2.3. As aimed, their inverse is indeed a reduction rule.

Theorem 2.12. *Algorithm 3 is a reduction rule, as defined in Subsection 1.1.*

Proof. Let us denote $f(D)$ the output of Algorithm 3 applied to a FDAG D. We need to prove that: (i) $f(D)$ is a subgraph of D and (ii) for any $D \neq D_0$, there exists an integer k such that $f^k(D) = D_0$, where D_0 is the FDAG with one vertex and no arcs.

(i) Since Algorithm 3 deletes either one vertex and its leaving arcs, or just one arc, $f(D)$ is indeed a subgraph of D.

(ii) The sequence of general term $f^k(D)$ is made of discrete objects whose size is strictly decreasing, therefore the sequence is finite and reaches D_0.

🪶

The associated expansion rule is exactly, in light of Proposition 2.11, the union of the three expansion rules (↕), (🌂) and (🏂). Since D_0, the DAG with one vertex and no arcs, is a FDAG, by virtue of what precedes and with Algorithm 1 – here with $g(\cdot) = \top$, we just defined an enumeration tree covering the whole set of FDAGs, whose root is D_0. A fraction of this enumeration tree is shown in Figure 7, illustrating the path from the root D_0 to the FDAG of Figure 3. Unexplored branches are ignored, but are still indicated by their respective root.

3 Growth of the tree

In this section, we analyse the enumeration tree defined in Section 2. In Subsection 3.1, we exhibit a bijection – Theorem 3.1 – between FDAGs and a class of combinatorial objects from the literature, allowing us to obtain an asymptotic expansion of the growth of the tree. In Subsection 3.2, we show that any FDAG has a linear number of children in that tree in Theorem 3.3, and that the time complexity to construct those children is quadratic – see Proposition 3.5. Finally, Theorem 3.6 states that our algorithm runs with polynomial delay [15].

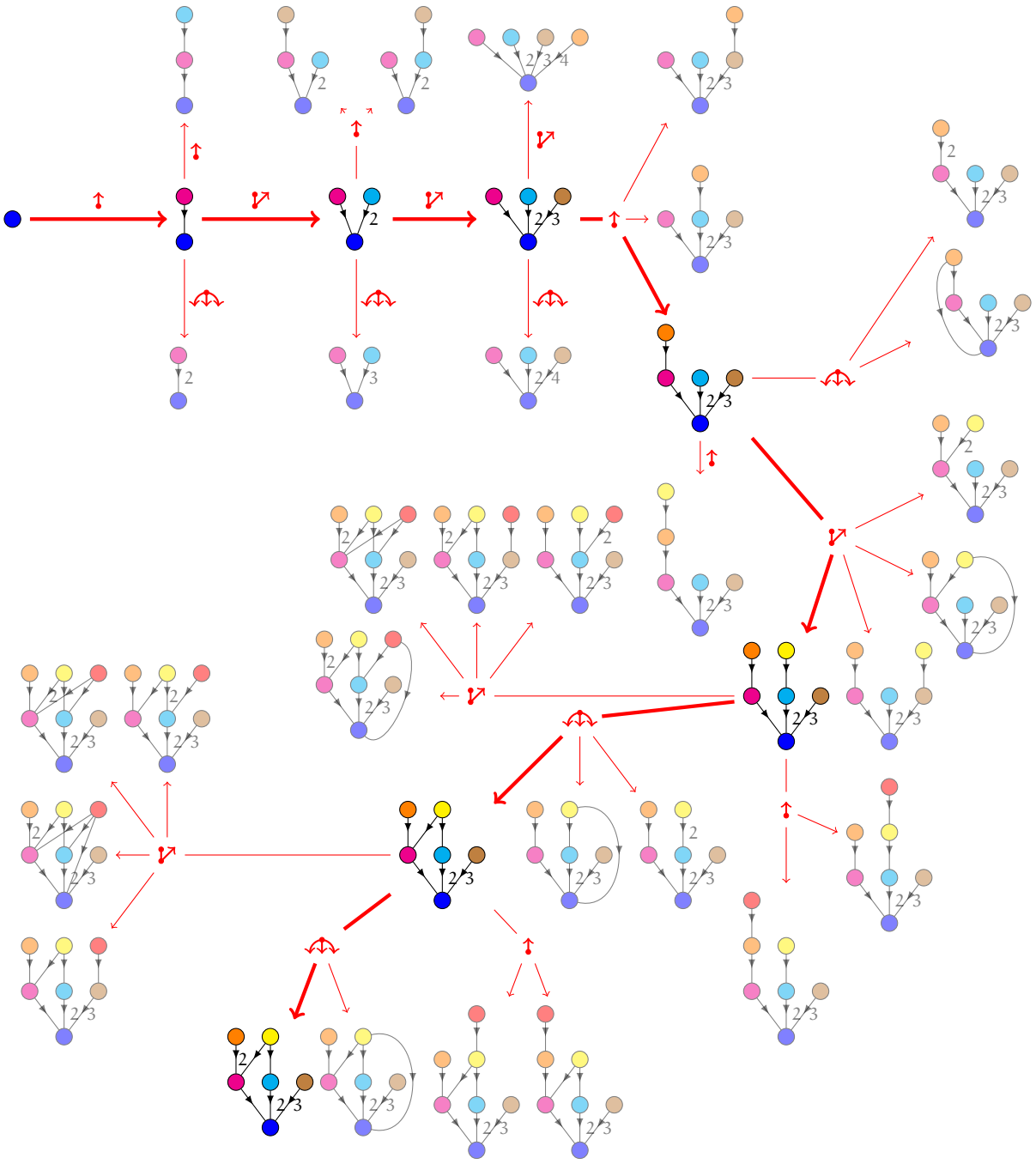


Figure 7: The path (in bold) in the FDAGs enumeration tree leading to the FDAG of Figure 3. The unexplored branches are only displayed by their root, which are shown partially transparent. The order of insertion of the vertices of each FDAG is always the same, and follows the color code (in the order of insertion): ●, ●, ●, ●, ●, ● and ●. With respect to the canonical ordering, they are numbered 0 to 6 in the same order.

3.1 Asymptotic growth

In this subsection, we show that FDAGs are in bijection with a set of particular matrices, whose combinatorial properties are known and give us access to an asymptotic expansion of the enumeration tree growth.

Let us denote E_k the set of all FDAGs that are accessible from D_0 in exactly k steps in the enumeration tree – with $\mathsf{E}_0 = \{\mathsf{D}_0\}$; then Table 2 depicts the values of $\#\mathsf{E}_k$ for the first nine values of k[3].

k	0	1	2	3	4	5	6	7	8
$\#\mathsf{E}_k$	1	1	3	12	61	380	2,815	24,213	237,348

Table 2: Number of FDAGs accessible from D_0 in k steps in the enumeration tree.

Actually, the terms of Table 2 coincide with the first terms of OEIS sequence A158691[4], which counts the number of *row-Fishburn matrices*, that are upper-triangular matrices with at least one nonzero entry in each row. The *size* of such a matrix is equal to the sum of its entries.

Theorem 3.1. *There exists a bijection Φ between the set of FDAGs and the set of row-Fishburn matrices, such that if D is a FDAG and $\mathsf{M} = \Phi(\mathsf{D})$, then*

$$\mathsf{D} \in \mathsf{E}_k \iff \mathit{size}(\mathsf{M}) = k.$$

Proof. The proof lies in Appendix A. 

This connection is to our advantage since Fishburn matrices (in general) are combinatorial objects widely explored in the literature as they are in bijection with many others – see [13, Section 2] for a general overview. Notably, the asymptotic expansion of the number of row-Fishburn matrices has been conjectured first by Jelínek [14] and then proved by Bringmann et al. [7].

Proposition 3.2 (Jelínek, Bringmann et al.)**.** *As $k \to \infty$,*

$$\#\mathsf{E}_k = k! \left(\frac{12}{\pi^2}\right)^k \left(\beta + O\left(\frac{1}{k}\right)\right)$$

with $\beta = \frac{6\sqrt{2}}{\pi^2} e^{\pi^2/24} = 1.29706861206\ldots$.

3.2 Branching factor

Given the overall structure of FDAGs, it is no surprise that the enumeration tree grows extremely fast. However, despite this combinatorial explosion, we show in this subsection that the branching factor, i.e., the outdegree of the nodes in the enumeration tree, is controlled. Actually, we prove that any FDAG has a linear number of successors[5] in the enumeration tree.

Theorem 3.3. *Any FDAG D has $\Theta(\#\mathsf{D})$ successors in the FDAG enumeration tree.*

Proof. Let $\mathsf{D} = (v_0, \ldots, v_n)$ be a FDAG. We denote $\mathcal{C}_\psi(v_n) = a_0 \cdots a_m$. Depending on the rule chosen:

[3] These numbers were obtained numerically (cf. "Implementation" at the end of the article).
[4] OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences, http://oeis.org/A158691.
[5] "successor" in the sense of "children in the enumeration tree". We make the distinction to avoid confusion with the children denoted by $\mathcal{C}(\cdot)$.

(↗↘) a_{m+1} belongs to $\mathcal{A}_< = \{0, \dots, p\}$, so the maximum number of successors is at most $p+1$, and at least 1, depending on the condition $a_m \geq_{\text{lex.}} a_{m+1}$.

(↕) The child of the new vertex is taken from $\mathcal{A}_= = \{p+1, \dots, n\}$ so the number of successors is exactly $n-p$.

(↗) Following Proposition 2.3, the number of successors is exactly $\#\mathcal{A}_< = p+1$.

Combining everything, the number of successors is at least $n+2$ and at most $n+p+2 \leq 2n+1$ (as $p \leq n-1$, with equality for FDAGs obtained just after using (↕) rule).

In the previous proof, we have shown that the number of successors of a FDAG with n vertices is between $n+1$ and $2n-1$. Figure 8 illustrates that these boundaries are tight, on 1 000 randomly generated FDAGs. A random FDAG is constructed as follows.

Definition 3.4 (Random FDAG). *Let $k \geq 0$. Starting from D_0 – the root, construct iteratively D_i as a successor of D_{i-1} in the enumeration tree, picked uniformly at random. We stop after k steps, and keep D_k.*

In Figure 8, we have generated 10 random FDAGs for each $k \in \{1, \dots, 100\}$.

It is indeed a suitable property that any FDAG admits a linear number of successors; but it would be of little use if the time required to compute those successors is too important. We demonstrate in the following proposition that temporal complexity is manageable. There are two possible strategies: (i) one can keep the enumeration tree in memory, and store on each node only the increment allowing to construct a FDAG from its predecessor; or (ii) one can explicitly build the successors by copying the starting FDAG, so that the tree can be forgotten. Depending on whether one wants to build the tree itself or only the FDAGs that compose it, one will choose either strategy.

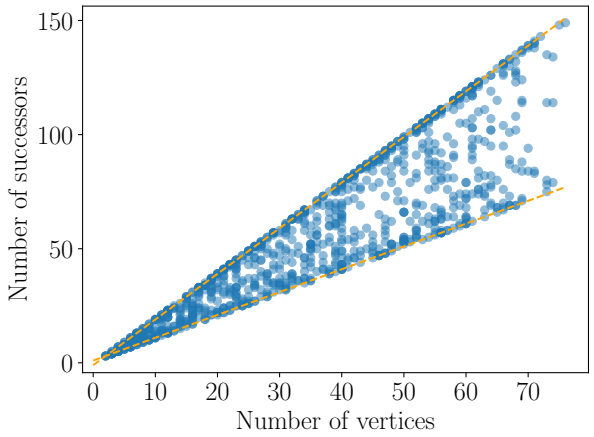


Figure 8: Numbers of successors of 1,000 random FDAGs in the enumeration tree, according to their number of vertices. Orange lines have equations $y = n+1$ and $y = 2n-1$.

Proposition 3.5. *Computing the successors of any FDAG D has complexity:*

(i) $O(\#D \deg(D))$ *if the construction is incremental from* D*;*

(ii) $O\left((\#D \deg(D))^2\right)$ *if the construction involves copying* D*.*

Proof. Let $D = (v_0, \dots, v_n)$ be a FDAG with $n+1$ vertices, with $\mathcal{C}_\psi(v_n) = a_0 \cdots a_m$, $\mathcal{A}_< = \{0, \dots, p\}$ and $\mathcal{A}_= = \{p+1, \dots, n\}$. Although the alphabets $\mathcal{A}_<$ and $\mathcal{A}_=$ can be retrieved in linear time, it is more efficient to maintain the pair (n, p) during enumeration; how to update these indices has already been presented in Subsection 2.3, when introducing each expansion rule.

The explicit construction of the successors in case (ii) requires to copy the vertices of D and their children, leading to a complexity in the order of $\sum_{i=0}^{n}(1 + \deg(v_i))$, which can be roughly bounded by $(n+1)(\deg(D)+1)$.

Depending on the expansion rule, the complexity for computing the new vertex or new arc varies:

- (i) The last letter of $\mathcal{C}_\psi(v_n)$ determines the number of successors – but it is no more than $p+1$. In case (i), although we could just store the information of the new letter, it is better to copy $\mathcal{C}_\psi(v_n)$ and add the new letter and store the result. Indeed, this allows to always have the knowledge of $\mathcal{C}_\psi(v_n)$ in the enumeration tree. The complexity for case (i) is therefore bounded by $\deg(v_n)(p+1)$; whereas it is $p+1$ in case (ii) since $\mathcal{C}_\psi(v_n)$ is already copied.
- (ii) Each successor is obtained by picking one element of $\mathcal{A}_= = \{p+1, \dots, n\}$. The complexity is exactly (up to a constant) $n-p$ in both cases.
- (iii) The successors are obtained by Algorithm 2, involving copying subwords of $\mathcal{C}_\psi(v_n)$ – the overall complexity is bounded by $(p+1)\deg(v_n)$.

The overall complexity is therefore of the order of $2(p+1)\deg(v_n)+n-p$ in case (i) and of $(n+1)(\deg(D)+1)\,[(p+1)(\deg(v_n)+1)+n-p]$ in case (ii). Using rough bounds, with $\deg(v_n) \leq \deg(D)$ and $p \leq n$, we end up with the stated complexity.

Whereas Figure 8 shows the number of successors of 1,000 random FDAGs, we measured the time needed to compute *explicitly* – i.e., implying copy, which is case (ii) in the previous Proposition 3.5 – these successors. The results are depicted in Figure 9, where we plotted (in blue) the total time t_D for computing all successors of a given FDAG D, and (in red) what we call *amortized time*, i.e. $t_D/(\#D\,\deg(D))^2$. As expected from Proposition 3.5, one can observe an asymptotic quadratic behaviour for the total time (in blue); concerning amortized time (in red), despite some variability, the upper bound seems to be constant.

The computations have been made on a MacBook Pro (2014) with an Intel Core i7 2.8 GHz processor and 16 GB of RAM.

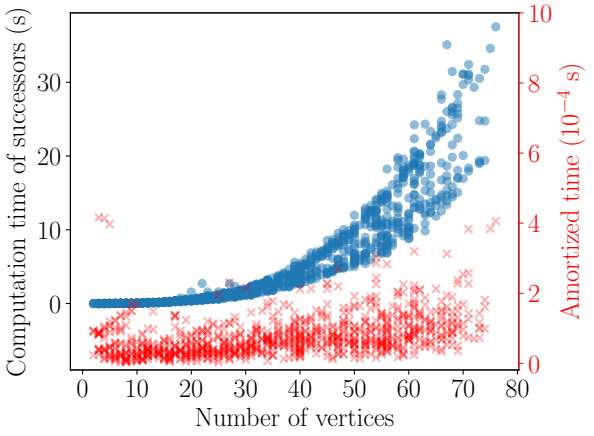


Figure 9: Total computation time (in blue) and amortized time (in red) for the explicit construction of the successors of the 1,000 random FDAGs of Figure 8, according to their number of vertices.

3.3 Polynomial delay

Let $E_{\leq K+1} = \bigcup_{k \leq K+1} E_k$ be the set of all FDAGs reachable in at most $K+1$ steps from the root of the FDAG enumeration tree. In this subsection, we show that the time complexity for enumerating $E_{\leq K+1}$ can be expressed as a function of the cardinality of $E_{\leq K}$ and has polynomial delay.

Theorem 3.6. *Enumerating* $E_{\leq K+1}$ *has time complexity* $O(K^2 \# E_{\leq K})$.

Proof. We adopt the configuration where we keep the enumeration tree in memory and where each node contains the incremental information to construct a FDAG from its predecessor.

We first observe the following: as D_0, the root, has one vertex and no arcs, and since the rules of expansion can only add one vertex and/or increase the degree of the last vertex by one, for any $D \in E_k$, it follows naturally that $\#D \leq k+1$ and $\deg(D) \leq k$. It implies that the time complexity for generating the successors of a FDAG in E_k is $O(k^2)$, according to case (i) of Proposition 3.5.

Thus, enumerating all the elements of E_{k+1} requires a time complexity of $O(k^2 \#E_k)$. It follows that we have a complexity of $O\left(\sum_{k \leq K} k^2 \#E_k\right)$ for enumerating $E_{\leq K+1}$. Since $k \leq K$ and $\sum_{k \leq K} \#E_K = \#E_{\leq K}$, we end up with the announced complexity.

As such, our algorithm has a polynomial delay, which is desirable for this kind of enumeration [15].

4 Variations on the enumeration tree

In this section, two variants of the enumeration tree presented in Section 2 are introduced. Subsection 4.1 proposes a way to enumerate forests in their classical sense, i.e., where redundancies within the forest are accepted, by adding an extra step following the previous enumeration. Finally, options to constrain the enumeration tree – on maximum number of vertices, height or outdegree – and making it finite are proposed in Subsection 4.2.

4.1 Extension to forests with repetitions

The enumeration tree constructed in Section 2 only allows to enumerate, in their compressed form, irredundant forests, where no tree can be a subtree of (or equal to) another. In this subsection, we propose a method to enumerate forests in the usual sense, without this non-redundancy restriction.

Let F be a forest in the classical sense, i.e., where some trees may be identical to or subtrees of other trees. If we compute $D = \mathfrak{R}(F)$, by definition, all these redundancies will be lost: the trees which are subtrees of another will be compressed with these subtrees in the obtained DAG, and those which are identical will be compressed in the same source of the DAG. This is why we specified in Subsection 1.2 that DAG compression is lossless if and only if the forest is irredundant.

We can preserve the information lost by the compression if we keep, in addition, a *presence vector*. Let us rewrite $D = (v_0, \ldots, v_n)$ according to the canonical order. Each tree $T \in F$ is associated with an index $i \in \{0, \ldots, n\}$ such that $T = \mathfrak{R}^{-1}(D[v_i])$. The presence vector $\pi_F : \{0, \ldots, n\} \to \mathbb{N}$ is constructed such that $\pi_F(i)$ counts how many times the tree $\mathfrak{R}^{-1}(D[v_i])$ appears in F. Thus, the couple (D, π_f) completely characterizes the forest F. To enumerate all (redundant) forests, it is therefore sufficient to enumerate both all FDAGs (corresponding to irredundant forests) and the presence vectors that may be associated with them.

Let D be an FDAG constructed in the FDAG enumeration tree. We define π_D as the presence vector associated to the (irredundant) forest $\mathfrak{R}^{-1}(D)$. This vector can be computed in a linear traversal of D, where the sources of D are assigned a value of 1 and the other vertices are assigned a value of 0. Adding redundancies in a forest means incrementing the presence vector, each $+1$ resulting in a new tree, whether it is equal to an existing tree or a subtree of it.

Our strategy is to enumerate, from π_D, all presence vectors corresponding to forests whose DAG reduction would be exactly D. To do so, we use a reverse search structure, with the following expansion rule (E). Let j be the index of the last increment, initialized to $j = 0$.

Definition 4.1 (E). *Choose any index* $j' \geq j$. *Increase* $\pi(j')$ *by one and set* $j \leftarrow j'$.

This rule allows to get any presence vector from π_D in a unique way, i.e., each index must be increased to its desired final value before moving to the next index. This defines an enumeration tree of presence vectors. If we implement this tree in such a way that each node contains only the new index j', we obtain an algorithm that enumerates each presence vector from its parent in constant time and space. The growth of this (infinite) new enumeration tree is given by the following proposition.

Proposition 4.2. *The number of redundant forests that can be constructed in at most* $k \geq 1$ *steps from* $\mathfrak{R}^{-1}(D)$ *– following expansion rule* (E) *– is given by* $\binom{n+1+k}{k} - 1$.

Proof. We first notice that the expected number is exactly the same as the number of presence vectors constructible in at most k steps from π_D. We then notice that if the current node (in the presence vector enumeration tree) has index j, then it has $n + 1 - j$ successors by the expansion rule (E) of Definition 4.1. For instance, since the starting index is 0, for $k = 1$, we obtain the indices $0, \dots, n$ in one copy each. We denote by $n_p(j)$ the number of times the index j appears in the nodes obtained in exactly p steps from the origin. Thus, $n_1(j) = 1$ by the above. Each index $j' \leq j$ existing at step $p - 1$ will induce a successor with index j at step p, so that $n_p(j) = \sum_{j'=0}^{j} n_{p-1}(j')$.

We establish by induction on p that $n_p(j) = \binom{k-1+j}{j}$, using the so called hockey-stick identity $\sum_{r=0}^{m} \binom{n+r}{r} = \binom{n+1+m}{m}$ [16]. Since the number of presence vectors that can be constructed in at most $k \geq 1$ steps from π_D is given by $\sum_{p=1}^{k} \sum_{j=0}^{n} n_p(j)$, we obtain the expected result after applying twice the hockey-stick identity. ∎

We can merge the enumeration tree of repetitions with the enumeration tree of FDAGs, to form a single enumeration tree, which enumerates forests in the classical sense (and in compressed form), as follows: the nodes of the enumeration tree carry a couple (FDAG, presence vector), and the available expansion rules are (↶↷), (↕), (↱↳) and (E). However, successors created with the last rule produce branches where it becomes the only rule available. In other words, once one chooses repetition, one can not modify any longer the topology of the FDAG – this is to ensure that each forest can only be enumerated in a unique way.

4.2 Constraining the enumeration

In [20], the authors propose an algorithm to enumerate all trees with at most n vertices. They simply check whether the current tree has n vertices or not, and as their expansion rule adds one vertex at a time, they decide to cut a branch in the enumeration tree once they have reached n vertices. Similarly, adding a vertex to a tree can only increase its height or outdegree, so we can proceed in the same way to enumerate all trees with maximal height H and maximal outdegree d. Indeed, the number of trees satisfying those constraints is finite [4, Appendix D.2].

This property also holds with the approach presented in Section 2: following one of the three expansion rules, we can only increase the height, outdegree or number of vertices of the FDAG. So, it makes sense to define similar constraints on the enumeration. However, for this constrained enumeration to generate a finite number of FDAGs, constraints must be chosen wisely, as shown in the following proposition.

Proposition 4.3. *The enumeration tree of FDAGs is finite if at least one of those set of constraints is chosen:*

(i) maximum number of vertices n *and maximum outdegree* d*,*

(ii) maximum height H *and maximum outdegree* d*.*

Proof. As (↶) allows to add arcs indefinitely without changing the numbers of vertices, constraining on the maximum outdegree is mandatory in both cases. As the two others rules add vertices, constraining by the number of vertices leads to a finite enumeration tree – (i) is proved. To conclude, we only need to prove that (↱) can not be repeated an infinite number of times, i.e. there is only a finite number of new vertices that can be added at a given height, up to the maximum outdegree. This is achieved by virtue of the upcoming lemma.

Let $H > 2$ and $d \geq 1$. Let D be the FDAG constructed so that for each $0 \leq h \leq H$, D has the maximum possible number n_h of vertices of height h and with maximum outdegree d. Initial values are $n_0 = 1$ and $n_1 = d$.

Lemma 4.4. $\forall 2 \leq h \leq H, \quad n_h = \sum_{k=1}^{d} \binom{k + n_{h-1} - 1}{k} \binom{d - k + n_0 + \cdots + n_{h-2}}{d-k}.$

Let $h \geq 2$ be fixed. To lighten the notation, let $n = n_{h-1}$ and $m = n_0 + \cdots + n_{h-2}$. Let v be a vertex to be added at height h. For any vertex v_i at height $h-1$, let x_i be the multiplicity of v_i in $\mathcal{C}(v)$ – 0 if $v_i \notin \mathcal{C}(v)$. Similarly, for any vertex v_h with $\mathcal{H}(v_j) \leq h-2$, y_j is the multiplicity of v_j in $\mathcal{C}(v)$ – possibly 0. By definition of $\mathcal{H}(\cdot)$ – see (1), at least one x_i is non-zero. Therefore, there exist $k \in [\![1, d]\!]$ such that:

$$x_1 + \cdots + x_n = k$$
$$y_1 + \cdots + y_m \leq d - k$$

By virtue of the stars and bars theorem, for a fixed k, there are $\binom{k+n-1}{k}$ choices for variables x_i, and $\binom{d-k+m}{d-k}$ for variables y_j. Summing upon all values for k proves the claim. ∎

Remark 4.5. *In the constrained enumeration proposed in [20], all the trees with* n *vertices are the leaves of the enumeration tree. To get all trees with* $n+1$ *vertices, it suffices to add to the enumeration all children of these leaves, i.e. trees obtained by adding a single vertex to them. This property – moving from one parameter value to the next by enumerating just one step further – does not hold anymore as soon as our set of constraints involve the maximum outdegree* d*, both for trees and FDAGs. For instance, from a FDAG of height* H*, one can obtain FDAG of height* $H+1$ *by using* (↕) *once and repeating* (↶) *up to* $d-1$ *times.*

5 Enumeration of forests of subtrees

Once the reverse search scheme has been set up to enumerate a certain type of structure, it is natural to move to a finer scale by using the same scheme to enumerate substructures. However, the notion of "substructure" is not obvious to derive from the main structure, as several choices are possible – e.g. for trees one can think of subtrees [24, 5], subset trees [8], etc. From a practical point of view, the enumeration of substructures permits to solve the frequent pattern mining problem – which will be tackled in Section 6.

In this section we define forests of subtrees, which will be our substructures. Compressed as FDAGs, these objects will be called subFDAGs. We then address the problem of enumerating all subFDAGs appearing in an FDAG D – similar as the one of enumerating all subtrees of a tree.

Forests of subtrees Similarly to forest being tuple of trees, *forests of subtrees* are tuple of subtrees, satisfying (2). Formally:

Definition 5.1. *Let* F *and* f *be two forests.* f *is a* forest of subtrees *of* F *if and only if*

$$\forall t \in f,\ \exists T \in F,\ t \in \mathcal{S}(T).$$

Forests of subtrees can be directly constructed from FDAGs, as shown by the upcoming proposition. Let D be a FDAG, and V be a subset of vertices of D.

Proposition 5.2. *If* $\forall v \in V$, $\mathcal{C}(v) \subseteq V$, *then* V *defines a FDAG* Δ, *such that* $\mathfrak{R}^{-1}(\Delta)$ *is a forest of subtrees of* $\mathfrak{R}^{-1}(D)$.

Proof. We recall from Subsection 1.2 that the notation $D[v]$ stands for the subDAG of D rooted in v composed of v and all its descendants $\mathcal{D}(v)$. The notation $\mathfrak{R}^{-1}(D[v])$ stands for the tree compressed by $D[v]$. The demonstration is in two steps. (i) Remove from D the vertices that does not belong to V; as there are no arcs that leave V by hypothesis, end up with a FDAG. Let us call Δ this FDAG. (ii) Let ρ be a root of Δ. By construction, ρ is also a vertex in D. Among all roots of D, there exists a root r such that $\rho \in \mathcal{D}(r)$. Therefore, $D[\rho]$ is a subDAG of $D[r]$, and then $t = \mathfrak{R}^{-1}(D[\rho])$ is a subtree of $T = \mathfrak{R}^{-1}(D[r])$ – with $T \in F = \mathfrak{R}^{-1}(D)$. As $\Delta[\rho]$ and $D[\rho]$ are isomorphic, $t \in f = \mathfrak{R}^{-1}(\Delta)$. Therefore we have proved that $\forall t \in f,\ \exists T \in F,\ t \in \mathcal{S}(T)$. ∎

We say that the FDAG Δ is a *subFDAG*[6] of D. Figure 10 provides an example of such a construction.

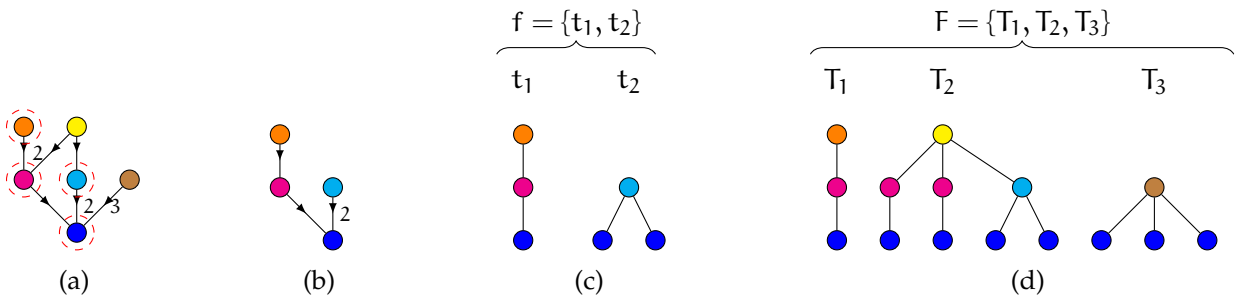


Figure 10: Construction of a forest of subtrees from FDAG. (a) A FDAG D. The set V is circled in red. (b) The FDAG Δ (c) The forest f compressed by Δ. (d) The forest F compressed by D. One can spot that $t_1 \in \mathcal{S}(T_1)$, $t_2 \in \mathcal{S}(T_2)$ so f is a forest of subtrees of F, and Δ a subFDAG of D.

[6]Not to be confused with *subDAG*, introduced in Subsection 1.2. A subDAG admits a single root and therefore compresses a single tree, whereas a subFDAG admits several roots and compresses a forest.

Enumeration of subFDAGs We now solve the following enumeration problem: given a forest F, find all forests of subtrees of F. Equally, given a FDAG D, find all subFDAGs of D. To address this, we make extensive use of the reverse search technique, adapting the one presented in Section 2.

Since a subFDAG is also a FDAG, it admits successors in the enumeration tree defined in Section 2. We are interested in those of these successors who are also subFDAGs (if any). In fact, since a subFDAG can be defined from a set of vertices, all one has to do is determine which new vertex can be chosen to expand an existing subFDAG – corresponding to a (↕) or (↗) step.The covering of all added new arcs is implicit in this construction and corresponds to some steps of (↷) .

Let Δ be a subFDAG of D and v its last inserted vertex – it is also the vertex with the largest ordering number in Δ. We denote by $S(\Delta)$ the set of all vertices $v' \in D$ that can be added to Δ to expand it to a new subFDAG. Let us call $S(\Delta)$ the *set of candidate vertices* of Δ. More precisely:

Lemma 5.3. *$S(\Delta)$ is the set of vertices $v' \in D$ that satisfies both:*

(i) $\mathcal{C}(v') \subseteq \Delta$

(ii) $\psi(v') > \psi(v)$

where $\psi(\cdot)$ is the canonical ordering of D.

Proof. (i) This condition is necessary so that $\Delta' = \Delta \cup \{v'\}$ fulfill the requirements for Proposition 5.2. (ii) This condition is necessary so that Δ' remains a FDAG. As $\psi(v') > \psi(v)$, either $\mathcal{H}(v') = \mathcal{H}(v)+1$ – then it is a (↕) step – or $\mathcal{H}(v') = \mathcal{H}(v)$ and $\mathcal{C}_\psi(v') >_{\text{lex.}} \mathcal{C}_\psi(v)$ – for a (↗) step. ∎

When $S(\Delta)$ is not empty, picking $s \in S(\Delta)$ ensure that $\Delta' = \Delta \cup \{s\}$ is a subFDAG of D. With respect to the enumeration tree of Section 2, Δ is an ancestor of Δ' – but not necessarily its parent, since the steps of (↷) are implicit. Δ' is called an *heir* of Δ. We can in turn calculate $S(\Delta')$, by updating $S(\Delta)$: (i) remove from $S(\Delta)$ all vertices v' such that $\psi(s) > \psi(v')$; (ii) in D, look only after the vertices v' such that $s \in \mathcal{C}(v') \subseteq \Delta \cup \{s\}$ and add them to $S(\Delta')$.

Algorithm 4: HEIRS

```
Input: D, [Δ, S(Δ)]
Set L to the empty list
for s ∈ S(Δ) do
    Let S' be a copy of S(Δ)
    S' ← S' \ {v' ∈ S' : C_ψ(v') ≤_lex. C_ψ(s)}
    S' ← S' ∪ {v' ∈ D : s ∈ C(v') ⊆ Δ ∪ {s}}
    Add [Δ ∪ {s}, S'] to L
return L
```

If Δ' is an heir of Δ, then by removing the last inserted vertex of Δ', one can retrieve Δ. This define a reduction rule f, and therefore an enumeration tree. Algorithm 4 is meant to construct the set $f^{-1}(\Delta)$. Applying Algorithm 1 together with it, and starting from $\Delta = \mathcal{L}(D)$[7] – in this case, $S(\Delta)$ is the set of parents of $\mathcal{L}(D)$ of height 1 – permits to enumerate all subFDAGs of D. Figure 11 provide an example by enumerating all subFDAGs of the FDAG of Figure 3.

6 Frequent subFDAG mining problem

Using the reverse search formalism defined in Subsection 1.1, the *frequent pattern mining problem* can be formulated as follows: from a dataset $X = \{s_1, \dots, s_n\}$ with $s_i \in \mathbb{S}$, and a fixed threshold σ, find

[7] where $\mathcal{L}(D)$ designates the leaf of D, i.e. the only vertex without children.

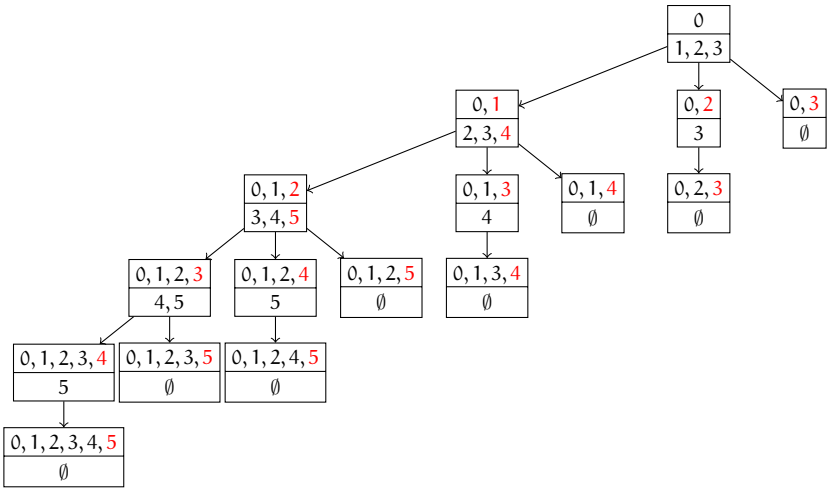


Figure 11: Enumeration tree of the subFDAGs of the FDAG of Figure 3, using both Algorithm 1 and Algorithm 4. The indices of the vertices correspond to the canonical ordering defined in Figure 3. In each vertex, the upper part corresponds to the current subFDAG Δ whereas the lower part stands for the set $S(\Delta)$. Numbers in red indicate what changes for an heir compared to its parent.

all elements $s \in \mathbb{S}$ that satisfy $\text{freq}(s, X) \geq \sigma$, where $\text{freq}(\cdot)$ is a function that counts the frequency of appearance of s in the dataset X. This problem can be solved using Algorithm 1 with the function $g(s, X, \sigma) = (\text{freq}(s, X) \geq \sigma)$, which is trivially anti-monotone.

We emphasize here that each possible definition of "s appears in X" leads to a different data mining problem. The choice of this definition is therefore of prime importance. In particular, this choice should induce a way of calculating $\text{freq}(s, X)$ that reflects the specificity of the reduction rule f, so that $\{s \in f^{-1}(s_0) | g(s) = \top\}$ can be constructed directly, instead of first generating $f^{-1}(s_0)$ and then filtering according to the value of g. Indeed, if g is too restrictive, and $f^{-1}(s_0)$ too large, one would have to enumerate objects that are not relevant to the enumeration problem, which is not desirable.

In this article, the problem we consider is the following: given a set of trees $X = \{T_1, \dots, T_n\}$, account for forests of subtrees that appear simultaneously in different T_i's. In other words, if we denote $\mathcal{F}_i$ the set of all forests of subtrees appearing in the forest formed by $\{T_i\}$, we are interested in the study of $\cap_{i \in I_\sigma} \mathcal{F}_i$ where $I_\sigma \subseteq [\![1, n]\!]$, such that $\#I_\sigma \geq \sigma \cdot n$.

A first, naive strategy would be to first build the $\mathcal{F}_i$'s, e.g. by using Algorithm 4 on $\mathfrak{R}(T_i)$, and then construct $\cap_{i \in I_\sigma} \mathcal{F}_i$ for all possible choices of I_σ. Obviously, this approach has its weaknesses: (i) many subFDAGs will be enumerated for nothing or in several copies, and (ii) it does not take into account that X is itself a forest. Our aim is to propose a variant of Algorithm 4 that, applied to $\mathfrak{R}(X)$, would enumerate only subFDAGs appearing in the $\mathfrak{R}(T_i)$'s with a large enough frequency.

Given a forest $F = \{T_1, \dots, T_n\}$ and its DAG compression $D = \mathfrak{R}(F)$, we have to retrieve, for each vertex in D, their origin in the dataset, that is, from which tree they come from. This issue has already been addressed in a previous article [5, Section 3.3], and has led to the concept of *origin*. For any vertex $v \in D$, the origin of v is defined as

$$o(v) = \left\{ i \in [\![1, n]\!] : \mathfrak{R}^{-1}(D[v]) \in \mathcal{S}(T_i) \right\},$$

where $\mathfrak{R}^{-1}(D[v])$ designates the tree compressed by the subDAG $D[v]$ rooted in v. In other words, $o(v)$ represents the set of trees for which $\mathfrak{R}^{-1}(D[v])$ is a subtree. We state in [5, Proposition 3.4] that origins can be iteratively computed in one exploration of D. The proof lies in the property that if $i \in o(v)$, then for all $v' \in \mathcal{D}(v)$, $i \in o(v')$ – as $\mathfrak{R}^{-1}(D[v']) \in \mathcal{S}(\mathfrak{R}^{-1}(D[v]))$.

Let Δ be a subFDAG of D. For Δ to compress a forest of subtrees of a tree T_i, it is necessary that $i \in o(v)$ for all $v \in \Delta$. Therefore, the set of trees for which Δ compress a forest of subtrees – the *origin* of Δ, denoted by $\mathbb{o}(\Delta)$ – is equal to

$$\mathbb{o}(\Delta) = \bigcap_{v \in \Delta} o(v).$$

If $\Delta' = \Delta \cup \{s\}$ is an heir of Δ – as defined earlier, then $\mathbb{o}(\Delta') = \mathbb{o}(\Delta) \cap o(s)$. Algorithm 4 can therefore be refined so that Δ' should be ignored if $\mathbb{o}(\Delta') = \emptyset$ – as Δ' does not anymore compresses any forest of subtrees actually present in the trees of F.

So far we neglected the threshold σ. We only want to keep subFDAGs that appear in at least σ% of the data. If $\#\mathbb{o}(\Delta)/\#F < \sigma$, then the successors of Δ are not investigated. Indeed, as $\mathbb{o}(\cdot)$ is a decreasing function, successors of Δ can not exceed the threshold again.

We can finally introduce Algorithm 5 that solves the frequent subFDAG mining problem for trees. With the notations of Subsection 1.1, this algorithm builds the set $\{\Delta' \in f^{-1}(\Delta) | \text{freq}(\Delta', F) \geq \sigma\}$, with $\text{freq}(\Delta', F) = \#\mathbb{o}(\Delta')/\#F$. The set is also built directly, without any posterior filtering, which is suitable as discussed at the beginning of the present subsection.

Algorithm 5: FREQUENTHEIRS

Input: $D = \mathfrak{R}(F)$, $[\Delta, S(\Delta), \mathbb{o}(\Delta)]$, σ

Set L to the empty list
for $s \in S(\Delta)$ **do**
  if $\mathbb{o}(\Delta) \cap o(s) \neq \emptyset$ **and** $\#(\mathbb{o}(\Delta) \cap o(s)) \geq \sigma \cdot \#F$ **then**
    Let S' be a copy of $S(\Delta)$
    $S' \leftarrow S' \setminus \{v' \in S' : \mathcal{C}_\psi(v') \leq_{\text{lex.}} \mathcal{C}_\psi(s)\}$
    $S' \leftarrow S' \cup \{v' \in D : s \in \mathcal{C}(v') \subseteq D_0 \cup \{s\}\}$
    Add $[\Delta \cup \{s\}, S', \mathbb{o}(\Delta) \cap o(s)]$ to L
return L

We stated earlier that we wanted to avoid generating unnecessary or multiple copies of subFDAGs, which is achieved with Algorithm 5. We now empirically study what we have gained from this, by comparing the use of Algorithm 5 on $D = \mathfrak{R}(F)$, with the use of Algorithm 4 on each $\mathfrak{R}(T_i)$. As in Subsection 3.2, we generated 1 000 random FDAGs D_k, 10 repetitions for each $k \in \{1, \ldots, 100\}$, creating D_k as in Definition 3.4. We assume $D_k = \mathfrak{R}(f)$ where $f = \{\mathfrak{R}^{-1}(D_k[r]) : r \text{ source of } D_k\}$. For each D_k, we have computed the quotient

$$Q(D_k) = \frac{\text{number of subFDAGs of } D_k \text{ enumerated via Algorithm 5}}{\sum_{r \text{ source of } D_k} \text{number of subFDAGs of } D_k[r] \text{ enumerated via Algorithm 4}}$$

with parameter $\sigma = 0$ when using Algorithm 5. The results are provided in Figure 12. Despite a rather marked variability, there is a general trend of decreasing as the number of vertices increases. We obtain fairly low quotients, around 20%, quite quickly. Given the combinatorial explosion of the objects to be enumerated, such an advantage is of the greatest interest.

Implementation

The `treex` library for Python [3] is designed to manipulate rooted trees, with a lot of diversity (ordered or not, labeled or not). It offers options for random generation, visualization, edit operations, conversions to other formats, and various algorithms. The enumeration of Section 2 and the algorithms of Section 5 and 6 have been implemented as a module of `treex` so that the interested reader can manipulate the concepts discussed in this paper in a ready-to-use manner. Installing instructions and the documentation of `treex` can be found from [3].

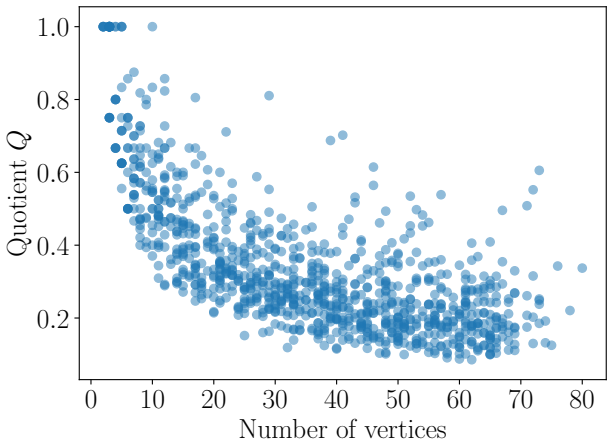


Figure 12: Quotient $Q(D)$ according to the number of vertices of D. Here 1 000 random FDAGs are displayed.

Acknowledgments

This work has been supported by the European Union's H2020 project ROMI. The authors would like to thank Dr. Arnaud Mary for his helpful suggestions on the draft of the article. Finally, the authors would like to thank two anonymous reviewers for their valuable comments, which help them to significantly improve the quality of their manuscript.

References

[1] Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin-Ichi Nakano. Discovering frequent substructures in large unordered trees. In *International Conference on Discovery Science*, pages 47–61. Springer, 2003.

[2] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

[3] Romain Azaïs, Guillaume Cerutti, Didier Gemmerlé, and Florian Ingels. treex: a python package for manipulating rooted trees. *The Journal of Open Source Software*, 4, 2019.

[4] Romain Azaïs, Jean-Baptiste Durand, and Christophe Godin. Approximation of trees by self-nested trees. In *ALENEX 2019 - Algorithm Engineering and Experiments*, pages 1–24, San Diego, United States, January 2019. URL: https://hal.archives-ouvertes.fr/hal-01294013, doi:10.10860.

[5] Romain Azaïs and Florian Ingels. The weight function in the subtree kernel is decisive. *Journal of Machine Learning Research*, 21:1–36, 2020.

[6] Edward A Bender and S Gill Williamson. *Lists, Decisions and Graphs*. S. Gill Williamson, 2010.

[7] Kathrin Bringmann, Yingkun Li, and Robert C Rhoades. Asymptotics for the number of row-fishburn matrices. *European Journal of Combinatorics*, 41:183–196, 2014.

[8] Michael Collins and Nigel Duffy. Convolution kernels for natural language. In *Advances in neural information processing systems*, pages 625–632, 2002.

[9] Philippe Flajolet and Robert Sedgewick. *Analytic combinatorics*. cambridge University press, 2009.

[10] Christophe Godin and Pascal Ferraro. Quantifying the degree of self-nestedness of trees: application to the structural analysis of plants. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 7(4):688–703, 2010.

[11] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery*, 15(1):55–86, 2007.

[12] John C. Hart and Thomas A. DeFanti. Efficient antialiased rendering of 3-d linear fractals. *SIGGRAPH Comput. Graph.*, 25(4):91–100, July 1991. URL: http://doi.acm.org/10.1145/127719.122728, doi:10.1145/127719.122728.

[13] Hsien-Kuei Hwang and Emma Yu Jin. Asymptotics and statistics on fishburn matrices and their generalizations. *arXiv preprint arXiv:1911.06690*, 2019.

[14] Vít Jelínek. Counting general and self-dual interval orders. *Journal of Combinatorial Theory, Series A*, 119(3):599–614, 2012.

[15] David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[16] Charles H Jones. Generalized hockey stick identities and n-dimensional blockwalking. 1994.

[17] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[18] Ailsa H Land and Alison G Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.

[19] Shin-Ichi Nakano. Efficient generation of plane trees. *Information Processing Letters*, 84(3):167–172, 2002.

[20] Shin-ichi Nakano and Takeaki Uno. Efficient generation of rooted trees. *National Institute for Informatics (Japan), Tech. Rep. NII-2003-005E*, 8, 2003.

[21] Sebastian Nowozin. *Learning with structured data: applications to computer vision*. PhD thesis, Berlin Institute of Technology, 2009.

[22] Benno Schwikowski and Ewald Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(1-3):253–265, 2002.

[23] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA, 1963. ACM. URL: http://doi.acm.org/10.1145/1461551.1461591, doi:10.1145/1461551.1461591.

[24] S.V.N. Vishwanathan and Alexander J Smola. Fast kernels on strings and trees. *Advances on Neural Information Proccessing Systems*, 14, 2002.

[25] Kazuaki Yamazaki, Toshiki Saitoh, Masashi Kiyomi, and Ryuhei Uehara. Enumeration of nonisomorphic interval graphs and nonisomorphic permutation graphs. *Theoretical Computer Science*, 806:310–322, 2020.

[26] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724. IEEE, 2002.

A A bijection between FDAGs and row-Fishburn matrices

This section is dedicated to the proof of Theorem 3.1, which is in two steps. First, we recall the natural bijection between FDAGs and their adjacency matrices; the latter are then put into bijection with the row-Fishburn matrices.

FDAG $\leftrightarrow$ Reduced adjacency matrix Let $D = (v_0, \dots, v_n)$ be a FDAG constructed in k steps from D_0 in the enumeration tree defined in Subsection 2.5. The adjacency matrix of D is defined as $A = (A_{i,j})_{i,j \in [\![n,0]\!]^2}$ where, if m is the multiplicity of v_j in $\mathcal{C}(v_i)$, then $A_{i,j} = m$ – possibly 0 if $v_j \notin \mathcal{C}(v_i)$. By construction of D, as v_n is the last inserted vertex, it has no parents, so $A_{n,\cdot}$ is a column of zeros; and as v_0 is a leaf, it has no children, so $A_{\cdot,0}$ is a row of zeros. We define the *reduced* adjacency matrix M as the matrix A deprived of this column and this row. Therefore, $M = (A_{i,j})_{i \in [\![n,1]\!], j \in [\![n-1,0]\!]}$. As a vertex can not be a parent to any vertex introduced after it, we have $A_{i,j} = 0$ for all $i \leq j$ – so that M is an upper-triangular matrix. In addition, as all vertices except v_0 have at least one child, there is at least one non-zero entry in each row of M. Therefore, M is a row-Fishburn matrix. However, we have no guarantee that this matrix verifies $\text{size}(M) = k$.

Reduced adjacency matrix $\rightarrow$ Incremental adjacency matrix Let $D = (v_0, \dots, v_n)$ be a FDAG, and M its reduced adjacency matrix. Let M_i be the row of M corresponding to $\mathcal{C}_\psi(v_i)$. The incremental adjacency matrix $\hat{M}$ is defined as:

$$\begin{cases} \hat{M}_1 = M_1 \\ \hat{M}_{i+1} = M_{i+1} \ominus M_i \end{cases}$$

where the $\ominus$ operation is defined as follow: given two rows $a_0 \cdots a_n$ and $b_0 \cdots b_n$, then denoting $j = \min\{i : a_i \neq b_i\}$, and $c = a_j - b_j$,

$$\begin{array}{rcccc} a_0 & \cdots & a_{j-1} & a_j & a_{j+1} \cdots a_n \\ \ominus\, b_0 & \cdots & b_{j-1} & b_j & b_{j+1} \cdots b_n \\ \hline = 0 & \cdots & 0 & c & a_{j+1} \cdots a_n \end{array} .$$

We claim that this new matrix $\hat{M}$ is a row-Fishburn matrix of size k, if $D \in E_k$. Actually, since M was already a row-Fishburn matrix, we just have to check that the size is correct. Let us consider v_i and v_{i+1}. The vertex v_{i+1} has been constructed from v_i by using either (↕) or (↗), and potentially several (↑↓) after that – let us say $p \geq 0$ times. Therefore, if the claim is correct, the sum over $\hat{M}_{i+1}$ should be exactly $p + 1$. Consider the operation by which v_{i+1} was added in the first place:

- (↕) $\mathcal{C}_\psi(v_{i+1})$ is reduced to a single element a, such that $a >_{\text{lex.}} \mathcal{C}_\psi(v_i)$. Therefore, the index j of the first non-zero coefficient of M_{i+1} is ahead of the one of M_i so that the coefficient c of $\ominus$ is equal to the j-th coefficient of M_{i+1} minus zero. Since the (↑↓) rule adds children to respect decreasing words, the p extra coefficients are added to the right of the j-th coefficient (including it) and therefore they are kept unchanged in the $\ominus$ operation. Eventually, the sum over M_{i+1} is $p + 1$ and so is the sum over $\hat{M}_{i+1}$.
- (↗) $\mathcal{C}_\psi(v_{i+1})$ is built from $\mathcal{C}_\psi(v_i)$ with Algorithm 2, and therefore they (i) share a common prefix, possibly empty and (ii) then differ by a single letter. The index of that letter in M_{i+1} corresponds to the index j defined in $\ominus$. Therefore, the coefficient c is – before any (↑↓) – equal to one. The argument of (↑↓) letters being added to the right of j still hold and therefore the sum over $\hat{M}_{i+1}$ is also $p + 1$.

To conclude the proof, we have to exhibit the inverse function of the mapping we just defined. This will prove that this mapping is indeed a bijection, and then the theorem holds.

Incremental adjacency matrix → Reduced adjacency matrix Let M and $\hat{M}$ be constructed as before. From $\hat{M}$, we can define a matrix M' as:

$$\begin{cases} M'_1 = \hat{M}_1 \\ M'_{i+1} = M'_i \oplus \hat{M}_{i+1} \end{cases}$$

where the $\oplus$ operation is defined as follow: given two words $a_0 \cdots a_n$ and $b_0 \cdots b_n$, then denoting $j = \min\{i : b_i \neq 0\}$, and $c = a_j + b_j$,

$$\begin{array}{rccc} & a_0 \cdots a_{j-1} & a_j & a_{j+1} \cdots a_n \\ \oplus & b_0 \cdots b_{j-1} & b_j & b_{j+1} \cdots b_n \\ \hline = & a_0 \cdots a_{j-1} & c & b_{j+1} \cdots b_n \end{array} .$$

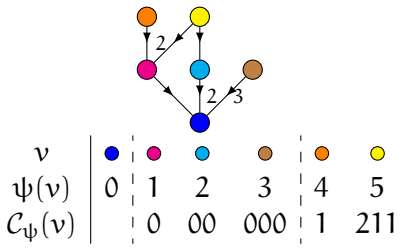
By construction, $\oplus$ is the inverse operation of $\ominus$, so that we have the following lemma:

Lemma A.1. *The following properties hold:*

- $M_i \oplus (M_{i+1} \ominus M_i) = M_{i+1}$
- $(M_i \oplus \hat{M}_{i+1}) \ominus M_i = \hat{M}_{i+1}$

Thefore, $M = M'$.

The FDAG of Figure 3 is reproduced below to illustrates the stages of the proof. This FDAG is constructed in 7 steps, that are (in this order): (↕) , (↱) , (↱) , (↕) , (↱) , (↶) and (↶) . The matrices A, M and $\hat{M}$ are given in Figure 13. One can see that $\hat{M}$ is of size 7, as expected.



v		v_5	v_4	v_3	v_2	v_1	v_0
$\psi(v)$	0	1	2	3	4	5	
$\mathcal{C}_\psi(v)$		0	00	000	1	211	

A	v_5	v_4	v_3	v_2	v_1	v_0
v_5	.	0	0	1	2	0
v_4	.	.	0	0	1	0
v_3	.	.	.	0	0	3
v_2	.	.	.	.	0	2
v_1	.	.	.	.	.	1
v_0	.	.	.	.	.	.

M	v_4	v_3	v_2	v_1	v_0
v_5	0	0	1	2	0
v_4	.	0	0	1	0
v_3	.	.	0	0	3
v_2	.	.	.	0	2
v_1	.	.	.	.	1

$\hat{M}$	v_4	v_3	v_2	v_1	v_0
v_5	0	0	1	2	0
v_4	.	0	0	1	0
v_3	.	.	0	0	1
v_2	.	.	.	0	1
v_1	.	.	.	.	1

Figure 13: The FDAG of Figure 3 reproduced (top left), its adjacency matrix A (top right), its reduced adjacency matrix M (bottom left) and its incremental adjacency matrix $\hat{M}$ (bottom right). Dots represent zeros corresponding to $A_{i,j}$ elements with $i \leq j$.

Remark A.2. *It should be noted that (general) Fishburn matrices, with at least one non-zero entry on each row and column, are in bijection with FDAGs compressing forests made of a unique tree. Indeed, via the bijection above, as such FDAG have a unique root, it must be the last inserted vertex, and therefore, each column admits at least one non-zero entry (otherwise it would be another root).*

Remark A.3. *It is possible to enumerate row-Fishburn matrices by using the previous bijection and the FDAGs enumeration tree together. Nevertheless, things are a little simpler in this case and the equivalent of the operations (↕) and (↱) can be merged, giving two rules for matrix expansion:*

(R1) Increase one coefficient to the (inclusive) left of the rightmost nonzero coefficient of the top row by 1.

(R2) Increase the dimension of the matrix by 1 *(to the left and top), all new coefficients set to zero. Set one coefficient of the top row to 1.*

B Index of frequent notations

Trees & DAGs v designates indifferently a vertex of a tree T or a DAG D.

$\mathcal{C}(v)$	*children* of v: all vertices connected to an arc leaving v
$\deg(v)$	*outdegree* of v: number of children of v
$\mathcal{D}(v)$	*descendants* of v: children of v, their children, and so on
$\mathcal{H}(v)$	*height* of v: length of the longest path from v to a leaf
$\#T, \#D$	number of vertices
$T[v], D[v]$	*subtree/subDAG* rooted in v and composed of v and $\mathcal{D}(v)$
$\mathcal{L}(T), \mathcal{L}(D)$	*leaves*: vertices without any children
$\deg(T), \deg(D)$	*outdegree*: maximum outdegree among all vertices
$\mathcal{S}(T)$	the set of all distinct subtrees of T
$\mathcal{T}$	the set of all trees
$\mathcal{F}$	the set of all *forests*, i.e. sets of trees such that no tree is a subtree of another

DAG reduction D designates a FDAG, F a forest.

$\mathfrak{R}(F)$	DAG reduction of the forest F
$\mathfrak{R}^{-1}(D)$	the forest F compressed by D, so that $\mathfrak{R}(F) = D$
$\mathfrak{R}^{-1}(D[v])$	the tree T compressed by $D[v]$, so that $\mathfrak{R}(\{T\}) = D[v]$

Canonical FDAGs Let D be a fixed FDAG, v any vertex of D and v_n the vertex with highest index in the canonical ordering.

$\psi(\cdot)$	canonical topological ordering of D
$\mathcal{C}_\psi(v)$	$\mathcal{C}(v)$ sorted by decreasing order on the indices defined by $\psi(\cdot)$
$\mathcal{A}_=$	the indices of all vertices of D with same height as v_n
$\mathcal{A}_<$	the indices of all vertices of D with strictly inferior height as v_n

$\Lambda_<$ the set of all *decreasing words* on $\mathcal{A}_<$, i.e. where each letter is greater than or equal to those who follow, w.r.t. the lexicographical order

$\Lambda_<^{\overline{w}}$ the set of all decreasing words *bounded by* $\overline{w}$, i.e. all words in $\Lambda_<$ that are greater than or equal to $\overline{w}$, w.r.t. the lexicographical order

$SC(w)$ *suffix-cut operator*: the word w deprived of its last letter

Enumeration tree

D_0 the FDAG with one vertex and no arcs

E_k the set of FDAGs that are accessible in exactly k steps from D_0 in the FDAGs enumeration tree

π_D *presence vector*: $\pi_D(i)$ counts how many times the tree $\mathfrak{R}^{-1}(D[v_i])$ appears in the forest $\mathfrak{R}^{-1}(D)$, for any FDAG $D = (v_0, \dots, v_n)$ and $i \in \{0, \dots, n\}$.

Forests of subtrees Let Δ be a subFDAG of D, and v any vertex. Let $F = \mathfrak{R}^{-1}(D)$.

$S(\Delta)$ *candidate vertices of* Δ: the set of all vertices v' of D so that $\Delta \cup \{v'\}$ is still a subFDAG of D

$o(v)$ *origin of* v: the set of indices i so that $\mathfrak{R}^{-1}(D[v])$ is a subtree of $T_i \in F$.

$\mathfrak{o}(\Delta)$ *origin of* Δ: the set of indices i so that $\mathfrak{R}^{-1}(\Delta)$ is a forest of subtrees of $T_i \in F$.