Entropic Uniform Sampling of Linear Extensions in Series-Parallel Posets[*]

Olivier Bodini[†], Matthieu Dien[‡], Antoine Genitrini[‡] and Frédéric Peschanski[‡].
`Olivier.Bodini@lipn.univ-paris13.fr` and
`{Matthieu.Dien,Antoine.Genitrini,Frederic.Peschanski}@lip6.fr`.

July 13, 2017

In this paper, we introduce a uniform random sampler for linear extensions of Series-Parallel posets. The algorithms we present ensure an essential property of random generation algorithms: entropy. They are in a sense optimal in their consumption of random bits.

1 Introduction

The state-space of a concurrent program is interpreted, in what is called the *interleaving semantics*, as the linearization of partially ordered sets (posets). This linearization process is highly combinatorial, a topic we studied thoroughly in previous papers. The uniform random generation of linear extensions provides a stochastic approach for the linearization process, hopefully avoiding the so-called "combinatorial explosion". For instance, in [1] we develop an efficient algorithm to draw linear extensions of tree-shaped posets. The algorithm has worst-case time complexity $O(n \log n)$ (counting arithmetic operations) with n the size of the poset (more precisely, the number of nodes of its covering tree). A uniform random sampler for posets of dimension 2 is introduced in [2]. A perfect sampling algorithm for arbitrary posets is presented in [4]. These are polynomial algorithms but in the order $\tilde{O}(n^3)$ hence not usable on large posets. Our goal is to identify subclasses of posets for which more efficient algorithms can be proposed. In this paper, we introduce a uniform random sampler for linear extensions of *Series-Parallel (SP) posets*. This represents a very important class of posets that can be found in many practical settings. Generating linear extensions uniformly at random for this subclass can be done in a relatively straightforward way. However, such a naive algorithm fails an essential property of random generation algorithms: *entropy*. When studying random generation, the consumption of random bits is a very important measure. An entropic algorithm minimizes this consumption, which has a major impact on efficiency. The algorithms we describe in the paper are *entropic*, i.e. they are in a sense optimal in their consumption of random bits.

The outline of the paper is as follows. First, in section 2 we define a canonical representation of Series-Parallel posets. Based on this representation we develop our random generation algorithms in section 3. We propose two variants of the random sampler: a bottom-up and a top-down variant. In section 4 we describe the common stochastic core of both algorithms. This is where we discuss, and prove, the property of *entropy*. In this extended abstract, we only provide outlines for the correctness and entropy proofs.

[*]This research is partially supported by the ANR project *MetACOnc*, ANR-15-CE40-0014.

[†]Laboratoire d'Informatique de Paris-Nord, CNRS UMR 7030 - Institut Galilée - Université Paris-Nord, 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse, France.

[‡]Laboratoire d'Informatique de Paris 6, CNRS UMR 7606 and Université Pierre et Marie Curie, 4 place Jussieu, 75005 Paris, France.

2 Canonical representation of Series-Parallel posets

The Series-Parallel posets are not easily handled from an algorithmic point of view. The ground set, the set of relations, and the inductive structure of a Series-Parallel poset must be adapted in order tu use them automatically through algorithms, see for example [11]. So, to work with Series-Parallel posets we introduce a canonical representation of such posets, based on their *covering* directed acyclic graph (DAG). Then we design an effective algorithm that can be precisely analyzed, thanks to the canonicity of our representation. In this section we detail such a DAG representation whose main objective is to preserve the necessary informations such that the uniformity property of the sampling process is preserved. We first recall the classical construction of Series-Parallel posets [10] (SP posets). This is based on the two basic composition rules below.

Definition 1 (Poset compositions)**.** *Let P and Q be two independent partial orders, i.e. posets whose ground sets are disjoint.*

- *The parallel composition R of P and Q, denoted by $R = P \parallel Q$, is the partial order obtained by the disjoint union of the ground sets of P and Q and such that*

$$\forall E \in \{P, Q\}, \forall x, y \in E, x \prec_R y \textit{ iff } x \prec_E y.$$

- *The series composition S of P and Q, denoted by $S = P.Q$, is the partial order obtained by the disjoint union of the ground sets of P and Q and such that*

$$\forall x, y \in P \cup Q, x \prec_S y \textit{ iff } \begin{cases} x \in P \textit{ and } y \in Q \\ x, y \in P \textit{ and } x \prec_P y \\ x, y \in Q \textit{ and } x \prec_Q y \end{cases}.$$

We are now ready for the definitions of SP posets.

Definition 2 (Series-Parallel partial orders)**.** *The class of Series-Parallel orders is the smallest class containing the atomic order and closed under series and parallel composition.*

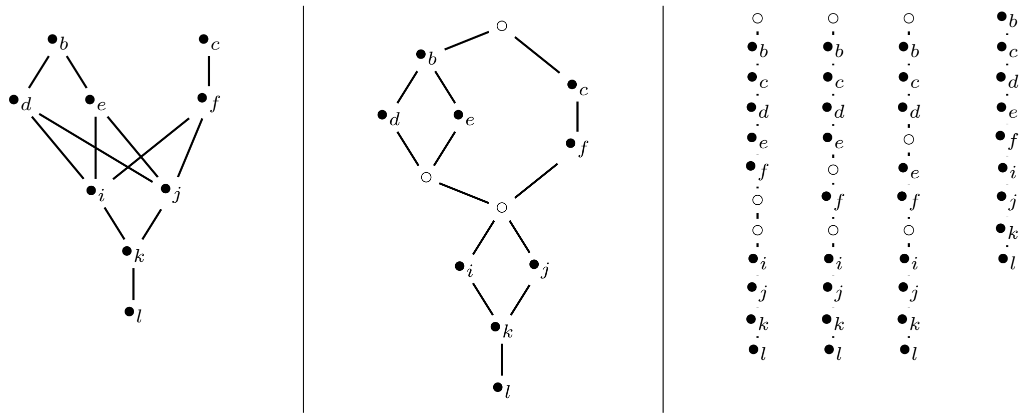

Figure 1: *(Left to Right) The covering of a Poset P; A SP DAG for P; The topological sorts of a given equivalence class and the associated linear extension.*

In the mathematical definition, a partial order is a set of relations between points of the ground set. We must adapt this context to obtain an efficient representation in computers. So, a common way to handle such a poset for an algorithmic need, is to exploit their *covering* DAG.

Definition 3. *Let P be a partial order. The covering G of P is the directed acyclic graph (DAG) whose nodes are the points of the ground set V of P and whose set of directed edges is such that*

$$\{(x, y) \in V^2 \mid (x \prec_P y) \land \neg(\exists z \in V,\ x \prec_P z \prec_P y)\}.$$

Note that the combinatorial class of coverings is the one of intransitive DAG.

The leftmost part of Fig. 1 represents the covering of a Series-Parallel poset. It represents the poset with ground set $\{b, c, d, e, f, i, j, k, l\}$. The set of order relation is $\{b < d, b < e, c < f, d < i, d < j, e < i, e < j, f < i, f < j, i < k, j < k, k < l\}$ and all transitive relations. We will use the common representation of coverings as Hasse diagrams in which edges are directed from top to bottom.

Definition 4. *Let P be a poset. A linear extension of P is a total order of the points of the ground set of P and satisfying the relations of P.*

The rightmost chain represented in Fig. 1, is the following linear extension of P: $b < c < d < e < f < i < j < k < l$. For the rest of the paper, the points of P will be called nodes. The following is *folklore.*

Fact 5. *Let P be a poset. Each linear extension of P corresponds to a topological sort of the nodes of the covering of P.*

We now introduce our special flavor of coverings, with the objective of getting effective and elegant algorithms presented in the next sections.

Let P a poset, and G its covering. First we want to distinguish the different children of a node in G. The classical way for this consists in a *combinatorial embedding* of G in the plane (cf. [5, Chapter 3])[1]. In our context, we choose an arbitrary embedding, e.g. in Fig. 1, we have chosen that b is on the left of c. Note that this arbitrary choice only impacts the representation and not the poset itself. Thus, we will identify the covering of a poset with the chosen combinatorial embedding.

A second simplification is that we only consider unary-binary DAGs instead of ones with nodes of arbitrary in-degree and out-degree. This is important otherwise an extra level of loop would be required in the algorithms. To reach this goal, we use the *left-leaning* principle [8] directly on the combinatorial embedding: a poset composed of several posets in parallel, associated to a combinatorial embedding of its covering, $P_1 \parallel P_2 \parallel \cdots \parallel P_n$ is seen as a poset with a binary parallel composition (relying on its associativity property): $(\ldots(P_1 \parallel P_2) \parallel \ldots) \parallel P_n$.

To encode the covering G with only unary-binary nodes, we need to introduce some "silent" nodes that we call *white nodes* in the rest of the paper, among the original *black nodes* of G (the points of the ground set of P). In the following, we explain the construction of this new structure. We also show how to recover the linear extensions of P from this representation.

Definition 6. *Let Ψ be the following function, from the set of combinatorial embeddings of Series Parallel poset coverings to the set of bicolored unary-binary (combinatorially embedded) DAGs. It is inductively defined as:*

$$\Psi(\varnothing) = \circ \qquad \Psi(\bullet) = \bullet \qquad \Psi\left(\begin{array}{c}\bullet\\ \vdots\\ \bullet\end{array}\right) = \begin{array}{c}\bullet\\ \vdots\\ \bullet\end{array}$$

$$\Psi(P.(Q \parallel R).S) \quad = \quad \begin{tikzcd} & \Psi(P) \arrow[dl, dash, dashed] \arrow[dr, dash, dashed] & \\ \Psi(Q) \arrow[dr, dash, dashed] & & \Psi(R) \arrow[dl, dash, dashed] \\ & \Psi(S) & \end{tikzcd},$$

where P, Q, R and S are arbitrary coverings of Series-Parallel posets. P and S may be empty. Q and R must not be empty and R must verify $\exists R_1, R_2,\ R = R_1 \parallel R_2$.

The Series-Parallel DAG (SP DAG) of a poset P is the image of an arbitrary combinatorial embedding of the covering of P by the function Ψ.

[1]The combinatorial embedding allows to distinguish the two successors of a node: the left one and the right one.

When the poset looks like $P_1 \parallel P_2 \parallel \cdots \parallel P_n$, the last pattern condition means $R = P_n$. Note that the unique constraint for P and S is that they are SP posets, eventually empty. In particular, they can contain substructure like $(A \parallel B)$. Thus the application of the last rule is not deterministic, in the sense that we can apply it sometimes successively, but the order of application is arbitrary. However the rules are trivially confluent.

In Fig. 1, on the left, a poset is represented by its covering. In the middle, it is the SP DAG we associate to this covering. A SP DAG contains bicolored nodes. The black nodes are the nodes of the initial covering, while the white nodes have been added to fulfill the unary-binary arity constraint.

The SP DAGs are by essence recursively decomposable, and thus we use the classical notation from Analytic Combinatorics [3] for their characterization.

Proposition 7. *The class $\mathcal{D}$ of SP DAGs is unambiguously specified by:*

$$\left\{\begin{array}{lcl}
\mathcal{D} & = & \bullet \quad + \quad \begin{array}{c}\bullet\\ | \\ \mathcal{D}_t\end{array} \quad + \quad \begin{array}{c}\bullet + \circ\\ \diagup \quad \diagdown\\ \mathcal{D} \qquad \mathcal{D}_r\\ \diagdown \quad \diagup\\ \mathcal{D} + \circ\end{array}\\[2em]
\mathcal{D}_t & = & \bullet \quad + \quad \begin{array}{c}\bullet\\ | \\ \mathcal{D}_t\end{array} \quad + \quad \begin{array}{c}\bullet\\ \diagup \quad \diagdown\\ \mathcal{D} \qquad \mathcal{D}_r\\ \diagdown \quad \diagup\\ \mathcal{D} + \circ\end{array}\\[2em]
\mathcal{D}_r & = & \mathcal{D}_t \quad + \quad \begin{array}{c}\circ\\ \diagup \quad \diagdown\\ \mathcal{D} \qquad \mathcal{D}_r\\ \diagdown \quad \diagup\\ \mathcal{D}\end{array}
\end{array}\right.$$

Let us recall the basic notation from Analytic Combinatorics, by describing the first equation. A DAG in $\mathcal{D}$ is either a single node $\bullet$ or a root $\bullet$ followed by a DAG from the class $\mathcal{D}_t$ or a *top* root (either a black node $\bullet$ or a white node $\circ$) with a *left* substructure belonging to $\mathcal{D}$ and a *right* substructure from $\mathcal{D}_r$, both followed by a *bottom* substructure corresponding either to a DAG in $\mathcal{D}$ or to a white node $\circ$. In the rest of the paper we will use the terms top, left, right and bottom substructures.

Remark that the class $\mathcal{D}_t$ contains the connected Series-Parallel coverings with a single source (i.e. a node smaller that all other nodes), and $\mathcal{D}_r$ encodes the class of connected Series-Parallel coverings.

Theorem 8. *For each Series-Parallel poset, we choose an arbitrary combinatorial embedding of the covering. Let $\mathcal{E}$ be the set of the chosen embeddings of the Series-Parallel posets. Ψ is a bijection from $\mathcal{E}$ to the set of SP DAGs.*

The proof is direct by a structural induction.

By successively using the combinatorial embedding, the left-leaning principle and the transformation Ψ, we have an effective and canonical way for representing Series-Parallel posets.

Another central property is the correspondence between the linear extensions of the poset P and the topological sorts of $\Psi(P)$.

Let P be a poset, and $S = \Psi(P)$ be its associated SP DAG. We consider the set of topological sorts of S. Obviously each of them contain all nodes of S, the black ones but also the white ones, although the latter have no meaning for P.

Definition 9. *Let P be a poset, and $S = \Psi(P)$ be its associated SP DAG. Let ρ be the function, from the set of topological sorts of S to the set of linear extensions of P, such that, applied to a topological sort all white nodes are removed.*

Definition 10. *Let P be a Series-Parallel poset, and $S = \Psi(P)$ be its associated SP DAG. We define the following equivalence relation based on the function ρ:*
Two topological sorts s_1 and s_2 of S are equivalent if and only if $\rho(s_1) = \rho(s_2)$.

Let us remark some fundamental constraint in the equivalence relation: for two distinct linear extensions of a poset, the numbers of sorts in the two corresponding equivalence classes are not necessary equal. In fact, in Fig. 1 (right-hand side), we have represented the three topological sorts corresponding to the same linear extension. But the following linear extension $c < f < b < d < e < i < j < k < l$, is given by a single topological sort, where both white nodes appear between e and i. As a consequence, we cannot directly sample uniformly a topological sort of S and then apply the transformation ρ to obtain *uniformly* a linear extension of P.

Definition 11. *Let P be a Series-Parallel poset, $S = \Psi(P)$ be its associated SP DAG. In an equivalence class of topological sorts of S, we define the representative to be the sort whose white nodes appear as soon as possible. If two white nodes are incomparable and appear successively in the representative we choose the leftmost one (in S) to appear first.*

In Fig. 1 (on the right side), among the three topological sorts, the rightmost one is considered to be the representative for the linear extension. The case where several white nodes are incomparable and successive in the topological sort is handled similarly.

Theorem 12. *Let P be a Series-Parallel poset, $S = \Psi(P)$ be its associated SP DAG. An uniform sampling of the linear extension of a Series-Parallel poset P is obtained by drawing uniformly at random a representative $\Psi(P)$ and by applying the function ρ to it.*

The proof of the theorem is a direct consequence of the previous results.
To conclude this section, we exhibit the computational complexity for the construction of the SP DAG resulting of a Series-Parallel poset.

Proposition 13. *The SP DAG corresponding to a Series-Parallel poset, whose ground set contains n nodes is built in $O(n)$ time complexity.*
The number of black nodes of the SP DAG is n and the number of white nodes is at most $2(n-1)$.

This last proposition guarantees that the complexity of building and using a SP DAG in place of its associated Series-Parallel poset, will be negligible in front of the complexity of the algorithms presented below.

3 Random generation of linear extensions

Based on the SP DAG structures defined in the previous section, we now begin the presentation of the uniform random samplers. In this section, we give the outline of the algorithms, and in the next section we discuss their common stochastic core. We present two complementary generation schemes: a *bottom-up* scheme that recursively generates the linear extensions "from the inside-out", and a *top-down* variant that does the converse. Both algorithms have an interest. The bottom-up approach is arguably the simplest, in that it follows the recursive decomposition of the input DAG. The correctness proofs are much easier in this setting. Comparatively, the top-down approach seems more convoluted, and it is best seen as a transformation of the bottom-up scheme in the proofs. But in practice the top-down algorithm is better in that it can be implemented *in-place*.

The bottom-up variant is Algorithm 1. We illustrate it on the poset example of Fig. 1. The root is an unlabeled white node, with two subposets with respective roots b and c. The join node is white and itself the root of the poset comprising the labels $\{i, j, k, l\}$. In the case of such a fork/join structure[2], the algorithm works as follows. First, the algorithm is recursively applied on the two subposets in parallel: the one with labels $\{b, d, e\}$ and the one with labels $\{c, f\}$. For the latter, there is only one possibility: taking first the label c and then f, resulting in the partial linear

[2]in reference to the Unix `fork` and `join` system calls to manage processes

Algorithm 1 Bottom-up variant of the uniform random generation of linear extensions

```
function RandLinExt-BU($P$)
    if $P = \circ$ then return [ ]
    else if $P = \bullet_x$ then return $[x]$
    else if $P = \bullet_x$ . $T$ then return cons(x, RandLinExt-BU($Q$))
    else if $P = \square$ . ($L$ ∥ $R$) . $T$ then
        $h$ := Shuffle(RandLinExt-BU($L$), RandLinExt-BU($R$))
        $t$ := RandLinExt-BU($T$)
        if $\square = \bullet_x$ then return concat(cons($x$, $h$), $t$)
        else return concat($h$, $t$)
```

extension $[c, f]$. For the left part, the label b is prepended to the *uniform shuffle* of the singleton linear extensions $[d]$ and $[e]$. The shuffle algorithm will be presented in detail, but in this situation it simply consists in taking either $[d, e]$ or $[e, d]$ both with a probability of $\frac{1}{2}$. Suppose we take the latter, we ultimately obtain the linear extension $[b, e, d]$. Note that nothing is appended at the end since the join node is white in this subposet. In the next step, the extensions $[c, f]$ and $[b, e, d]$ are shuffled uniformly, one possible outcome being $[c, b, e, d, f]$. This is then concatenated with a linear extension of the downward poset. For example $[c, b, e, d, f, i, j, k, l]$ is one such possibility, and is thus a possible output of the algorithm.

Algorithm 2 Top-down variant of the uniform random generation of linear extensions

```
function RandLinExt-TD($P$)
    function RecRandLinExt-TD($P$, $rankings$, $positions$)
        if $P = \circ$ then return $rankings$
        else if $P = \bullet_x$ then
            $rankings[x]$ := pop($positions$)
            return $rankings$
        else if $P = \bullet_x$ . $T$ then
            $rankings[x]$ := pop($positions$)
            return RecRandLinExt-TD($T$, $rankings$, $positions$)
        else if $P = \square$ . ($L$ | $R$) . $T$ then
            if $\square = \bullet_x$ then $rankings[x]$ := pop($positions$)
            $upPositions$ := $positions[\ 0\ \ldots\ |L| + |R| - 1\ ]$
            $botPositions$ := $positions[\ |L| + |R|\ \ldots\ |P| - 1\ ]$
            $l$, $r$ := Split($upPositions$, $|L|$, $|R|$)
            $rankings$ := RecRandLinExt-TD($L$, $rankings$, $l$)
            $rankings$ := RecRandLinExt-TD($R$, $rankings$, $r$)
            return RecRandLinExt-TD($T$, $rankings$, $botPositions$)
    $rankings$ := an empty dictionary
    $positions$ := $[\ 1\ \ldots\ |P|\ ]$
    return RecRandLinExt-TD($P$, $rankings$, $positions$)
```

The top-down variant is described by Algorithm 2. The main difference with the bottom-up algorithm is that it samples *positions* in an array, instead of *labels* directly. The advantage is that most operations can then be performed in-place, at the price of having to deal with a level of indirection. The *rankings* structure is a mapping associating the node labels to a position in the sampled linear extension. The *positions* are organized as a stack structure, initially containing all the available positions from 1 to $|P|$ (the size of the poset in the number of labels i.e. black nodes). For our example poset the initial contents of *positions* is $[1, 2, 3, 4, 5, 6, 7, 8, 9]$. The *rankings* map is empty. In the first step, the white root is simply skipped and the two sets of positions are computed: the *upPositions* taking the front part of the poset i.e. $[1, 2, 3, 4, 5]$ and *botPositions* what is remaining i.e. $[6,7,8,9]$. The algorithm then performs an *uniform split* of the positions 1 to 5 e.g. in a subset $l = \{2,3,4\}$ for $\{b,d,e\}$ and $r = \{1,5\}$ for $\{c,f\}$. The details about the splitting process are given below. The rankings of each subposet are computed recursively, and the result naturally interleaves since we work with disjoint sets of positions. Once again, we can ultimately obtain the linear extension $[c,b,e,d,f,i,j,k,l]$. We have to show that it is obtained with the same exact (uniform) probability as in the bottom-up case.

Algorithm 3 Algorithm of uniform random splitting and shuffling

```
function SPLIT($S$, $p$, $q$)
    $\ell$, $r$ := [ ], [ ]
    $i$ := 0
    $v$ := RandomCombination($p$, $q$)
    for all $e \in v$ do
        if $e$ then
            append $S[i]$ to $\ell$
        else
            append $S[i]$ to $r$
    return $\ell$, $r$
```

```
function SHUFFLE($\ell$, $r$)
    $t$ := [ ]
    $v$ := RandomCombination($|\ell|$, $|r|$)
    for all $e \in v$ do
        if $e$ then
            append pop($\ell$) to $t$
        else
            append pop($r$) to $t$
    return $t$
```

In fact, those two algorithms are dual. In the bottom-up case, randomness comes from the Shuffle function which is the dual of the Split function in the sense of a coproduct. A shuffle takes two lists and mixes them into one, while the split takes one list and divides it into two. The key property comes from the fact that the shuffle (resp. split) of one (resp. two) lists is sampled uniformly: each shuffle (resp. splits) has the same probability to be drawn. For example, there is $\binom{5}{2} = 10$ possible shuffles between the sets $\{a,b,c\}$ and $\{d,e\}$. Equivalently there is 10 possibles splits of the set $\{a,b,c,d,e\}$ into two subsets, one of size 3 and the other of size 2.
Both algorithms operate in the same way: they draw a random combination of p elements among $p+q$, then shuffle or split using this combination. This will be discussed in the next section. Based on the assumption that the stochastic process is uniform, we obtain a first important result about the algorithms.

Theorem 14. *Algorithm 1 and Algorithm 2 both generate a linear extension of a series-parallel poset P uniformly at random. Their worst-case time complexity is $\Theta(n^2)$ (by measuring the number of memory writings). The average time complexity is equivalent to $\frac{1}{4}\sqrt{\frac{\pi\, n^3}{3\sqrt{2}-4}}$.*

Fact 15. *(Möhring [7]). Let P be a SP poset and ℓ_P be its number of linear extensions. If $P = P_1 \times P_2$ is the series composition of P_1 and P_2, then $\ell_P = \ell_{P_1} \cdot \ell_{P_2}$. If $P = P_1 + P_2$ is the parallel composition of P_1 and P_2, then $\ell_P = \binom{n_1+n_2}{n_1} \cdot \ell_{P_1} \cdot \ell_{P_2}$, where n_1 (resp. n_2) is the size of P_1 (resp. P_2).*

sketch. The correctness of both algorithms is easily proved by a structural induction and by using the Fact 15.

To compute the average complexity, the idea is to find a recurrence equation for the number of size n SP DAGs and another recurrence equation for counting the number of cumulated memory writings on size n SP DAGs. Thus, using standard analytic combinatorics tools [3], we derive the asymptotic behaviors of the solutions of both recurrences. The result is obtained by dividing the asymptotic number of cumulated memory writings by the one of size n SP DAGs. □

4 Entropic sampling core

The bottom-up and top-down algorithms described in the previous section both depend upon the same *stochastic core*: namely the procedure we named RandomCombination. Random generation must adopt the point of view of probabilistic Turing machines, i.e. deterministic machines with a tape containing random bits. As a consequence, an important measure of complexity for such an algorithm is the entropy (information theoretic meaning [9]) of the targeted distribution: the number of random bits consumed to produce a possible output. Our objective is to define *entropic* algorithms, according to the following definition.

Definition 16 (Entropic algorithm)**.** *Let A be an algorithm sampling an element of a finite set S at random according to a probability distribution μ. We say that A is* entropic *if the average number of random bits n_e it uses to sample one element $e \in S$ is proportional to the entropy of μ, in the sense of Shannon's entropy:*

$$\exists K > 0, \forall e \in S, n_e \leqslant K \cdot \sum_{x \in S} -\mu(x) \log_2(\mu(x)).$$

The key idea in the following entropic algorithms is to show that Bernoulli random variable (i.e. r.v.) of small or big parameter has weak entropy and because we are unable to use fraction of bits we group it in packs to draw Bernoulli r.v. of large entropy *i.e.* 1. The maximum entropy of a Bernoulli is reached when the parameter is $\frac{1}{2}$, in this case the Bernoulli r.v. is just a random bit.

Algorithm 4 Algorithm of uniform random generation of combination

```
function RandomCombination($p$, $q$)
    $l := [\,]$
    $\sharp$True is the number of True in $l$
    $\sharp$False is the number of False in $l$
    $rndBits$ := a stream of random booleans produced with $k$-Bernoulli$\left(\frac{p}{p+q}\right)$
    if $p > \log(q)^2 \wedge q > \log(p)^2$ then
        while $\sharp$True $<= p \wedge \sharp$False $<= q$ do
            if pop($rndBits$) then $l$ := cons(True, $l$)
            else $l$ := cons(False, $l$)
        $remaining := \neg$pop($l$)
    else
        if $p < q$ then
            $l$ := a list of $q$ times False
            $remaining$ := True
        else
            $l$ := a list of $p$ times True
            $remaining$ := False
    for $i := \sharp$True + $\sharp$False $- 1$ to $p + q - 1$ do
        $j$ := uniformRandomInt$[0 \dots i]$
        insert $remaining$ at position $j$ in $l$
    return $l$
```

The core of the random samplers is presented in Algorithm 4. The objective is to draw, in an entropic way, a list l of booleans of size $p + q$ such that p cells contains a True value and the q remaining are set to False.

We give an example of the sampling process for $p = 6$ and $q = 2$. In the first step, the list l is filled with True and False values, with respective probability $\frac{p}{p+q}$ and $\frac{q}{p+q}$. For this we use a stream of Bernoulli random variables *rndBits* that is produced by a function named k-Bernoulli, which we explain below. The filling process stops when one of the boolean values is drawn once more than needed (e.g. reaching $p + 1$ (resp. $q + 1$) times True (resp. False) values). The last value is then discarded. For example, if l = F :: T :: T :: F :: T :: F :: T :: [ ] then F is drawn 3 times although only 2 is needed. So the last F is discarded. In the second step of the algorithm, a number *remaining* of boolean values is needed to complete the list l. These are randomly inserted among the bits already drawn, by using uniform integer random variables. For example:

$$\begin{aligned} & l = \mathtt{T} :: \mathtt{T} :: \mathtt{F} :: \mathtt{T} :: \mathtt{F} :: \mathtt{T} :: [\,] \\ \hookrightarrow\ & l = \mathtt{T} :: \mathtt{T} :: \mathtt{F} :: \underline{\mathtt{T}} :: \mathtt{T} :: \mathtt{F} :: \mathtt{T} :: [\,] \\ \hookrightarrow\ & l = \mathtt{T} :: \mathtt{T} :: \mathtt{F} :: \mathtt{T} :: \mathtt{T} :: \mathtt{F} :: \mathtt{T} :: \underline{\mathtt{T}} :: [\,] \end{aligned}$$

At the end, the list l contains the required number of booleans values, and as we justify below, it is drawn uniformly.

An important part of the algorithm is the first test $p > \log(q)^2 \wedge q > \log(p)^2$. This tests if p is largely smaller than q (or q largely smaller than p). In this case we skip the Bernoulli drawing step to directly insert the smallest number of booleans in the bigger one. This particular case is due to a change of rate in the distribution of the binomial coefficient.

Theorem 17. *The* RandomCombination *algorithm uniformly samples a list of p `True` and q `False`. It uses an entropic number of random bits.*

sketch. Let l be a drawn list of p `True` and q `False`. The correctness proof, is divided into two cases:

- l was drawn entirely during the first step: in this case the probability to draw l is directly the product of the probability to draw each boolean *i.e.* $\left(\frac{p}{p+q}\right)^p \left(\frac{q}{p+q}\right)^q$ and because this probability does not depend of l it is the same for each l
- l was drawn after $p+q-k$ Bernoulli samples: using the previous argument, this combination of $p+q-k$ values is uniformly drawn. Then each remaining boolean is uniformly inserted, and so, each combination of $p+q-k+1$ to $p+q$ booleans is uniformly built from the previous one

Concerning the random bit efficiency of the algorithm, we assume that k-Bernoulli is entropic, which we will establish later. Assuming this, the main idea of the proof is to analyze the number of consumed booleans in the stream *rndBits*.

To do this we let the random variable T to be the sum of $\sharp$`True` and $\sharp$`False` at the end of the first step (when the list l if filled initially). Thus, we have

$$\mathbb{P}(T = t) = \binom{t}{p}\left(\frac{p}{p+q}\right)^{p+1}\left(\frac{q}{p+q}\right)^{t-p} + \binom{t}{q}\left(\frac{p}{p+q}\right)^{q-t}\left(\frac{q}{p+q}\right)^{q+1}$$

Then, we compute the expected value T and get $\mathbb{E}[T] = (p+q) + o(p+q)$ when $p > \log(q)^2$ (resp. $q > \log(p)^2$). The latter condition justifies the first test of the algorithm. It remains to count the number of random bits used and to compare it to the entropy of the combinations of p among $p+q$ elements *i.e.* the entropy of $\binom{p}{p+q}$.

We recall that a uniform integer between 0 and n can be drawn with $\mathcal{O}(\log n)$ random bits. We let $B_{p,q}$ the number of random bits used to sample a Bernoulli random variable of parameter $\frac{p}{p+q}$. We recall that the entropy of such variable is $-p \log \frac{p}{p+q} - q \log \frac{q}{p+q}$.

Thus, the number of average random bit used is $\mathbb{E}[T]B_{p,q} + o(p+q)\mathcal{O}(\log n)$. The $o(p+q)\mathcal{O}(\log n)$ term come from the second step of the algorithm *i.e.* the uniform insertions of remaining booleans. To conclude, the average number of random bits used is asymptotically equal to $(p+q)B_{p,q}$ which is equivalent to the entropy of $\binom{p}{p+q}$. $\square$ $\square$

The entropic property of the RandomCombination relies on the entropy of the k-Bernoulli function, which is described by Algorithm 5. The key idea is to draw Bernoulli random variables of parameter p by packs, using the fact that a successful Bernoulli r.v. of parameter p^k corresponds to a sequence of k successes of a Bernoulli r.v. of parameter p. Thus, the parameter $k = \left\lfloor \frac{\log \frac{1}{2}}{\log p} \right\rfloor$ is such that p^k is close to $\frac{1}{2}$. This allow to draw Bernoulli r.v. of parameter close to $\frac{1}{2}$, for which our Bernoulli is entropic. Let us consider the following example of a call to k-Bernoulli with the argument $\frac{2}{7}$. In that case we let $p = 1 - \frac{2}{7} = \frac{5}{7}$ and so $k = 2$. Then we present the different possible runs in the form of a decision tree. We draw a Bernoulli r.v. of parameter $\left(\frac{5}{7}\right)^2$ and:

Algorithm 5 Sampling of k Bernoulli random variables

```
function k-Bernoulli($p$)                                    ▷ $p$ is less than 1
    function k-BernoulliAux($p$)
        $k := \left\lfloor \frac{\log \frac{1}{2}}{\log p} \right\rfloor, \ i := 0$
        $v :=$ a vector of $k$ times True
        while ¬Bernoulli($\sum_{\ell=0}^{i} \binom{k}{\ell} p^{k-\ell}(1-p)^{\ell}$) do
            $i := i + 1$
            $j :=$ uniformRandomInt($[0 \ldots k-1]$)
            $v[j] :=$ False
        return $v$
    if $p < \frac{1}{2}$ then return negate(k-BernoulliAux($1-p$))
    else return k-BernoulliAux($p$)
```

- if the Bernoulli draw is successful then two successes of Bernoulli r.v. of parameter $\frac{2}{7}$ is returned
- else, it is a fail, it means that at least one of the two Bernoulli r.v. of parameter $\frac{2}{7}$ is a fail, which has a probability $2 \cdot \frac{5}{7} \cdot \frac{2}{7}$ to happen, so we need to redraw a new r.v.
  - if it is successful, this means that only one variable is a fail and so we need decide which one
  - else, we need to draw one more r.v. of parameter 1, in other words a successful r.v., and we return two failed r.v.

The last brick of this framework is the Bernoulli algorithm, already known in the litterature (as explained by [6]).

Algorithm 6 Sampling of Bernoulli random variable

```
function Bernoulli($p$)                                      ▷ $p$ is less than 1
    function RecBernoulli($a$, $b$, $p$)
        if RandomBit() = 0 then
            if $m > p$ then return False
            else RecBernoulli($\frac{a+b}{2}$, $b$, $p$)
        else
            if $m < p$ then return True
            else RecBernoulli($a$, $\frac{a+b}{2}$, $p$)
    return RecBernoulli(0, 1, $p$)
```

Theorem 18. *The Algorithm 6 draws a Bernoulli random variable of parameter p using, in average,* 2 *random bits.*

Proof. The correction proof is direct. Just remark that if we note K the number of calls to the recursive RecBernoulli function, K is equal to the length K prefix of the binary writings of p.

The average number of random bits used is the expectation of K:

$$\mathbb{E}[K] = \sum_{k=1}^{\infty} k \cdot \frac{1}{2}^k = \frac{1}{2} \cdot \left(\frac{\mathrm{d}}{\mathrm{dz}} \frac{1}{1-z} \right) \bigg|_{z=\frac{1}{2}} = 2$$

□ □

Theorem 19. *The Algorithm 5 draws* $\left\lfloor \frac{\log \frac{1}{2}}{\log p} \right\rfloor$ *Bernoulli random variable of parameter p entropically.*

sketch. Let N be the number of iteration of the **while** loop in the k-Bernoulli algorithm, we get that the number of random bits used is upper bounded by $2N + N \log k$: the number of bits used to draw N Bernoulli plus the number of bits used for the N uniform (over $[0 \dots k]$) r.v. draws.

So, the expected number of random bits used is

$$\sum_{n=0}^{k} \binom{k}{n} p^{k-n}(1-p)^n(2(n+1) + n\log k) = 2 + (1-p)(2 + \log k)$$

We have to average it by the number of Bernoulli r.v. drawn this way *i.e.* k. So, the average number of random bits used to draw one Bernoulli r.v. of parameter p is $\frac{2}{k} + (1-p)(2+\log k)$. The minimum of this function in k is reached when $k = \frac{2\log 2}{1-p}$, in other word when the average number of random bits used is greater or equal to 2. This corresponds to the case where $k = 1$ in the algorithm k-Bernoulli: in this case we should directly use Bernoulli. In the other case, we obtain that the average number of random bits used is entropic. ▫ □

References

[1] Bodini, O., Genitrini, A., Peschanski, F.: A Quantitative Study of Pure Parallel Processes. Electronic Journal of Combinatorics 23(1), P1.11, 39 pages (2016)

[2] Felsner, S., Wernisch, L.: Markov chains for linear extensions, the two-dimensional case. In: SODA. pp. 239–247. Citeseer (1997)

[3] Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)

[4] Huber, M.: Fast perfect sampling from linear extensions. Discrete Mathematics 306(4), 420–428 (2006)

[5] Klein, P., Mozes, S.: Optimization Algorithms for Planar Graphs (To appear)

[6] Lumbroso, J.: Optimal discrete uniform generation from coin flips, and applications. CoRR abs/1304.1916 (2013), `http://arxiv.org/abs/1304.1916`

[7] Möhring, R.H.: Computationally Tractable Classes of Ordered Sets. Institut für Ökonometrie und Operations Research: Report (1987)

[8] Sedgewick, R.: Left-leaning red-black trees. In: Dagstuhl Workshop on Data Structures. p. 17 (2008)

[9] Shannon, C.E.: A mathematical theory of communication. ACM SIGMOBILE Mobile Computing and Communications Review 5(1), 3–55 (2001)

[10] Stanley, R.P.: Enumerative Combinatorics: Volume 1. Cambridge University Press, New York, NY, USA, 2nd edn. (2011)

[11] Valdes, J., Tarjan, R.E., Lawler, E.L.: The recognition of series parallel digraphs. In: Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing. pp. 1–12. STOC '79, ACM, New York, NY, USA (1979), `http://doi.acm.org/10.1145/800135.804393`