Quantitative and algorithmic analysis of concurrent programs

Martin Pépin

▶ To cite this version:

Martin Pépin. Quantitative and algorithmic analysis of concurrent programs. Discrete Mathematics [cs.DM]. Sorbonne Université, 2021. English. NNT : 2021SORUS450 . tel-03722845

HAL Id: tel-03722845
https://theses.hal.science/tel-03722845v1

Submitted on 13 Jul 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité : Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par
Martin Pépin

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Quantitative and algorithmic analysis of concurrent programs

Analyse quantitative et algorithmique des programmes concurrents

---

Soutenue le 29 septembre 2021 devant le jury composé de :

Olivier Bodini	Examinateur
Antoine Genitrini	Directeur de thèse
Manuel Kauers	Rapporteur
Frédéric Peschanski	Encadrant de thèse
Denis Poitrenaud	Examinateur
Viviane Pons	Examinatrice
Vlady Ravelomanana	Rapporteur
Christine Tasson	Examinatrice

Remerciements & Acknowledgements

I would like to start by warmly thanking my reviewers, Manuel Kauers and Vlady Ravelomanana, for accepting to read this manuscript in detail during their summer as well as my examiners, Olivier Bodini, Denis Poitrenaud, Viviane Pons, and Christine Tasson, for accepting to be part of my jury. I am honoured by the time and attention that you all put in my work.

Un énorme merci à vous deux, Antoine et Frédéric, pour m'avoir confié un sujet de recherche si passionnant, à la croisée des mathématiques et de l'informatique, pour tout votre soutien durant ces trois années, pour tous vos conseils, toute votre expérience, et tant de choses encore. Vous avez su me guider avec beaucoup de bienveillance dans tous les moments de la thèse et je vous suis infiniment reconnaissant pour ça.

Merci beaucoup à toute l'équipe APR et en particulier aux doctorant·es et post-doctorant·es. Ces trois années auraient été bien tristes sans vous et sans l'esprit de franche camaraderie qui règne au bureau 303. Merci à celles et ceux qui sont arrivé·es avant et aux post-docs pour leur expérience et leurs conseils : Marwan[1], Matthieu, Vincent, Alice, Ghiles, Clément, Steven, Boubacar, Yi-Ting, Pierre, Raouf. Merci à ma génération, Raphaël et Jules, pour cette aventure passée ensemble. Et enfin merci, et surtout bon courage, aux plus jeunes dont la constante bonne humeur illumine les journées au bureau : Mat(t)hieu, David, Francesco, Keanu et Guillaume.

I would also like to thank the ALEA community as a whole. I was astonished to discover such a welcoming and emulating group who accepted me as a member from the very first day. I have learnt so many things by discussing with you, ranging from new board games to elegant algorithmic tricks. En particulier, merci beaucoup à Olivier pour les heures passées au tableau à échanger des problèmes, merci à toi 2t pour tout le travail fait ensemble[2], pour m'avoir accueilli chez toi et pour avoir joué un rôle de grand frère pendant 3 ans. Merci à Mehdi pour les voyages faits ensembles. Many thanks to all the others: Isabella, Michael, Alex, Vincent, Sergey, Elie, Maciek. Merci à toute l'équipe de Caen pour m'avoir si souvent accueilli, et pour les pauses café[3] interminables.

Muchísimas gracias a ti Tuba, por el tiempo pasado en la Floresta, por las discusiones apasionantes sobre la investigación y sobre la vida. Y sobre todo, gracias por iniciarme al Catan!

[1] Oui tu comptes comme APR.

[2] Oui on va finir par l'écrire ce papier !

[3] La Figure 1.2 est pour vous.

Je voudrais terminer par remercier sincèrement les personnes qui ont été à mes côtés ces dernières années et qui les ont rendues riches et agréables. Merci aux « vieux » d'être là depuis si longtemps, merci pour les innombrables parties de coinche, de baby-foot, de pétanque, et pour les débats enflammés autour d'un verre. Merci à Niols pour ces deux années incroyables en coloc[4], pour nos nombreuses heures de TFA et surtout pour nos longues discussions qui ont tant aidé dans les moments difficiles et qui m'ont fait grandir. Merci à Kta pour avoir partagé un bout de chemin avec moi, pour m'avoir fait voyager et pour avoir écouté mes tracas de thésard avec une oreille bienveillante. Merci au A6 pour m'avoir ramassé à la petite cuillère et m'avoir accueilli dans les pires moments. Si je suis arrivé au bout de ce travail, c'est aussi grâce à vous, et à une rencontre en particulier qui a tout changé. Merci à ma famille pour votre soutien inconditionnel et pour la bulle protectrice que vous m'offrez, loin de la vie parisienne, quand j'ai besoin d'une pause.

En particulier merci à mon frère et à ma sœur, et ces derniers mots vous reviennent. Ces dernières années nous ont rapprochés et j'ai enfin compris que, bien que je sois le grand frère, c'est vous qui donnez les leçons de vie, et avec beaucoup de justesse. Merci.

[4]Pourquoi est-ce qu'on a arrêté la coloc déjà ?

Contents

Remerciements & Acknowledgements i

Contents iii

Introduction 1

Notations and generalities 6

1 Statistical analysis of non-deterministic fork-join programs 8
- 1.1 Context and related work . . . 8
- 1.2 A combinatorial interpretation of non-deterministic fork-join programs . . . 10
  - 1.2.1 Non-deterministic Fork-Join processes (without loops) . . . 10
  - 1.2.2 The combinatorial toolset part 1: modelling programs . . . 12
  - 1.2.3 The combinatorial toolset part 2: executions as partial increasing labellings 25
  - 1.2.4 Statistical analysis . . . 29
  - 1.2.5 Random sampling of executions . . . 31
  - 1.2.6 Experimental study . . . 34
- 1.3 Extending the model with loops . . . 35
  - 1.3.1 Non-deterministic fork-join processes with loops . . . 35
  - 1.3.2 Combinatorial interpretation . . . 37
  - 1.3.3 Statistical analysis algorithms . . . 41
  - 1.3.4 Experimental study . . . 48
- 1.4 Execution prefixes generation . . . 48
  - 1.4.1 Specification of the prefixes . . . 49
  - 1.4.2 Quantitative analysis . . . 51
  - 1.4.3 Uniform random sampling of prefixes . . . 55
  - 1.4.4 Experimental study . . . 56

2 Directed Ordered Acyclic Graphs and Multi-Graphs 59
- 2.1 Context and related work . . . 59
- 2.2 Directed Ordered Acyclic Graphs . . . 62
  - 2.2.1 Definition and recursive decomposition . . . 62
  - 2.2.2 Computational aspects of the enumeration . . . 64
  - 2.2.3 Counting and sampling algorithms . . . 66

2.2.4 Counting DOAGs by vertices only: asymptotic results . . . 69
2.2.5 Uniform sampling of DOAGs by vertices only . . . 74
2.3 Vertex-by-vertex decomposition of labelled DAGs . . . 76
2.3.1 Recursive decomposition . . . 76
2.3.2 Random generation . . . 77
2.4 Multi-graphs . . . 78
2.4.1 Edge-by-edge decomposition scheme . . . 79
2.4.2 Random sampling . . . 80
2.4.3 Bijection with decorated north-east paths . . . 80

3 Algorithmic considerations related to random generation 85
3.1 A review of random generation techniques . . . 86
3.1.1 Ad hoc algorithms and the rejection method . . . 86
3.1.2 Recursive random sampling . . . 87
3.1.3 Unranking (and ranking) . . . 91
3.1.4 Boltzmann sampling . . . 92
3.2 Faster lexicographic unranking of combinations . . . 96
3.2.1 Historical context and classical results . . . 96
3.2.2 Factoradics . . . 99
3.2.3 Realistic complexity analysis and algorithmic improvements . . . 103
3.3 Uniform random sampling of variations . . . 107
3.3.1 Definition and enumeration . . . 107
3.3.2 Random sampling . . . 108
3.3.3 Lexicographic unranking . . . 112
3.4 Usainboltz: fast and generic Boltzmann sampling . . . 118
3.4.1 Practical implementation . . . 118
3.4.2 Generic Boltzmann sampling with builders . . . 123

Conclusion 128
Contributions . . . 128
Perspectives . . . 129

Academic references 137

Technical references 143

Introduction

Numerous human activities are supported by software of increasing size and complexity. Over the years, concurrent programming has become an essential component of software development, but at the cost of more and more difficult programming abstractions. In concurrency, programs are not made of one linear sequence of actions any more. They are composed of several *processes* instead, which may execute their actions *sequentially*, independently from each other, create new processes, and synchronise with other processes. The order of execution of the actions of such a program is thus not fully specified, in particular when two or more processes are run in parallel, this is called *parallelism*. In practice, this parallelism can be implemented in hardware, this is the case on multi-core architectures, or in software, typically by a scheduler. These two kinds of parallelism serve not only a performance purpose (running independent tasks in parallel saves times) but also address an intrinsic need of nowadays computers (an operating system has to run the different programs of its potentially numerous users concurrently, handle IO communications, etc).

An important problem of concurrency is to ensure that such programs are correct. This question has been studied since the 60s in the context of concurrency theory [Pet62; Hoa78; Mil80]. Numerous techniques have been proposed to analyse concurrent programs (model-checking, statistical analysis, automated testing, etc.) but they all face the same problem of "combinatorial explosion" of the state-space or "state-explosion". This refers to the fact that the number of possible executions of such a program grows quickly with its size and this is due to the freedom in the order of execution of its actions. This thesis is part of a long-time project of quantitative analysis of this phenomenon combining viewpoints from concurrency and combinatorics. Our approach consists in modelling fundamental objects of concurrency as combinatorial classes so that we can analyse them using the tool-set of *analytic combinatorics* [FS09]. One aspect of this work is to study core program combinators, such as the parallel composition [BGP12; BGP16], the non-deterministic choice [BGP13] and some form of synchronisation [Die17; BDGP17a]. These notions were studied independently from each other in previous work and are integrated into a single unified framework in this thesis. An important notion underlying the structure of the state-space is that of partial orders. Partial orders offer good formalism for representing the control flow of concurrent programs since they allow to encode scheduling constraints (as ordering relations) without specifying the full execution order (as a total order). This approach has already been suggested in [BDGP19]. A second research direction we explore is to study the class of partial order as a whole by using directed acyclic graphs as a tractable approximation.

Tooling: the framework of (analytic) combinatorics

An essential aspect of our work is that we express concurrent problems in the framework of analytic combinatorics. The main interest of this framework is that it offers powerful tools to study precisely quantitative phenomenons such as the explosion of the state-space of programs, its typical shape or the number of syntactic programs themselves. A key component of analytic combinatorics is the use of generating functions, that is formal power series encoding integer sequences. They serve two purposes. First, they allow to express recurrence relations between these sequences in a concise and efficient way, thus allowing to perform high-level *computations* over them directly rather than reasoning in terms of recurrences. We use this encoding extensively in Chapter 1 to encode counting information on the state-space of programs and effectively implement counting algorithms. Second, these generating functions can be interpreted as functions of the complex variable whose analytic properties reflect the asymptotic behaviour of the initial sequences themselves. We also use this idea in Chapter 1 to obtain various quantitative results on the state-space of programs.

Another key idea of [FS09] is to describe the objects under study using a so-called "admissible" combinatorial *specification*. These specifications are context-free grammars augmented with a few operators specific to combinatorics, which give a high level description of the objects at play. In addition, this formalism allows to obtain equations satisfied by the relevant generating functions, in a systematic way, using a dictionary mapping each construction of the language of specifications to a functional operation. This connection between specifications and generating functions is at the heart of [FS09]. It allows to use techniques from complex analysis and forms a powerful framework to tackle the quantitative problems encountered in this thesis.

The field of combinatorics also provides a fertile ground for random generation which is omnipresent in our work. In particular, we use the systematic approach of [NW78] and [FZV94], called the "recursive method" to write many of the random samplers described in the thesis, though we need to adapt their complexity results to our slightly different paradigm. Another well-known random generation framework which we use here is that of Boltzmann sampling from [DFLS04]. This technique offers less control than the recursive method over the size of the generated objects but, in return, allows to generate larger structures.

Concurrency: from syntax to semantics

One of the goals of concurrency theory, and semantics in particular, is to give *syntactic* objects (i.e. programs) a meaning by mapping them to *semantic* objects (e.g. executions). We examine the two sides of this process in this work.

Semantics: analysing the state-space of concurrent programs First, we consider a class of programs combining the most fundamental operators of concurrency. We build on top of previous work where several key notions have been separately studied already, such as *parallelism*, seen as increasing labellings [BGP16], *non-determinism*, seen as partial labellings [BGP13], and *synchronisation* as constrained labellings. The starting point of our approach is to find a suitable combinatorial interpretation of these operators or, said differently, we express their *semantics* in combinatorial terms. A contribution of this thesis is to integrate these various interpretations, as

well as a new interpretation for loops, into a single *combinatorial specification*. We thus provide a unified framework for studying these operators and their interactions within a single language. This is a significant leap forward in terms of expressiveness, not only because of the presence of loops, but also because we integrate the non-determinism in a non-trivial language with synchronisation. A notable aspect of the formalism developed here is that it treats specifications as *dynamic* objects. That is to say that our algorithm builds a new specification at runtime for each input program. This differs from a common scenario in combinatorics where the focus is set on one specification and where solving a problem means analysing one particular combinatorial class. As a consequence, the size of the specifications we encounter becomes a parameter to be taken into account and forces us to refine some classical complexity results.

On the analytical side, expressing the semantics of programs as combinatorial classes allows us to study quantitative properties of their state-space. As an example, we quantify precisely the number of execution paths induced by the *choice* operator in the *average case*, thus giving a key witness of its expressiveness. Another example of such quantitative results is the study of the typical shape of the state-space of programs via the precise estimation of their number of executions prefixes.

At a more practical level, the framework we develop here is also an interesting source of algorithmic investigation, which is the main concern of this thesis. The first algorithmic problem we study is that of *counting* the number of executions (of bounded length, when in the presence of loops) of the programs. This is the question one has to answer to quantify the so-called *state-explosion* phenomenon, and this is an important building block of our algorithmic toolbox. Unfortunately, counting executions of concurrent programs is hard in the general case. It has been shown in [BDGP19] that, even for simple programs only allowing *barrier synchronisation*, counting executions is a #P-complete[5] problem. A second problem is caused by non-determinism because for each non-deterministic choice we have to select a unique branch of execution. Moreover, choices can be nested so that the number of possibilities can grow exponentially. In this thesis we opt for fork-join parallelism, which enables a good balance between tractability and expressiveness by enforcing some structure in the state-space. Moreover, another contribution is to provide an efficient encoding of the state-space as generating functions, allowing to count executions without expanding the choices. Of course counting executions has no direct practical application, but it is an essential requirement for us to build two complementary and more interesting analysis techniques. First, we develop a uniform random sampler of executions (again, of bounded length in the presence of loops). Without prior knowledge of the state-space, the uniform distribution yields the best coverage and thus offers a good default exploration strategy of the state-space. As a second and alternative approach, we provide a uniform random sampler of execution *prefixes*, offering a more flexible tool to explore the state-space of programs, while remaining in *control* over the distribution of the sampled objects. A fundamental characteristic of all the algorithms presented here is that they work on the *syntactic* representation of the programs, thus avoiding the explicit construction of the state-space. This allows to analyse systems with a large state space.

[5]A problem is in #P if it consists in counting the number of accepting paths of a polynomial-time non-deterministic Turing machine. It is #P-complete if, in addition, every other problem in #P reduces to it in polynomial time.

Syntax: DAGs as an approximation of programs A second and complementary aspect of this work is the study of syntactic objects. Here, rather than analysing the state-space of *one* particular program, we propose to dive into the class of programs themselves. A first step in this process is to define the class of programs to consider. A natural abstraction to model concurrent program is the set of partial orders. Partial orders are well suited to represent a set of atomic actions and a set of scheduling constraints on these actions, they encode the *control graph* of the program. This is the direction taken by [BDGP19] (in our line of work) where a decomposition of partial orders is proposed and used to count the number of possible executions of programs. Unfortunately, the decomposition is ambiguous and cannot be used in conjunction with analytic combinatorics tools to study the class of partial orders itself. This class is in fact hard to grasp combinatorially. A common simplification consists in approximating them by various classes of directed acyclic graphs (or DAGs) [JCB00; Cor+10; CEH19; 10], which thus provide an alternative encoding of the control graph. This motivates the second main contribution of this thesis which is to study DAGs and, in particular, a new class of DAGs called directed *ordered* acyclic graphs (or DOAGs).

The class of *labelled* DAGs, that is DAGs in which each vertex is identified by an integer, often presents itself as the default model when it comes to studying and generating DAGs. The presence of such a labelling is necessary to break symmetries which simplifies the analysis of these objects and, most importantly, makes their uniform random generation tractable. However, this labelling is not motivated by concurrency considerations or partial order theory. Our contribution is to introduce and study a new model of *unlabelled* DAGs (DOAGs) equipped with an ordering of their outgoing edges. We suggest this ordering as an alternative way to break symmetries. In terms of concurrency, this can be interpreted as ordering the sub-processes forked by a process. Although this is an information that one sometimes wants to ignore when modelling programs, this ordering does reflect a practical reality. In Unix for instance, this takes the form of a parent-child relationship: when a process forks, one of the two resulting execution threads is the parent and the other one is the child. Moreover, the bias introduced by this ordering favours DAGs with vertices of high degree. This has an important consequence: this bias is easier to grasp than the one introduced by labelling the vertices, and it can thus be balanced by bounding the out-degree of the graph, which we can easily achieve with our approach.

In order to count and sample DOAGs, we describe here a recursive decomposition of these objects which is of a different kind compared to previous approaches used in the DAG field (see [Rob73] and [Ges95] for instance). This decomposition allows us to describe an efficient uniform sampler of DOAGs offering a full control over the number of edges and vertices of the generated objects. We also show that the decomposition applied here can be adapted to recover some earlier results on the class of labelled DAGs and to enhance the state of the art in terms of random generation. In particular, we obtain new recursive formulas that are amenable to efficient random sampling with a *fixed* number of vertices and edges for the class of *labelled* DAGs.

Another, more exploratory, contribution of this thesis in the field of DAG theory is the study of a multi-graph variant of DOAGs. That is a model where several edges may appear between two given vertices. Using a refinement of the decomposition mentioned before, we manage to obtain an efficient recursive counting formula and an efficient uniform random sampler. This model, as well as the DOAG model also have applications outside of concurrency. For instance,

directed graphs with an ordering on the outgoing edges occur naturally in some applications such as the history of a file in the Git version control system [8, page 17]. Another natural use case of such objects is for modelling the memory layout of partially compacted tree-like structures. For instance, it is common in functional programming to share some common subtrees of persistent data-structures to save space. This technique is called *hash-consing* [Ers58; Got74]. For this particular use case, the multi-graph model is more suited since it allows to encode more partially compacted trees than the DOAG model.

Practical algorithmic considerations regarding random generation

In addition to providing quantitative results and algorithmic solutions to the problems exposed above, studying concurrency under the lens of combinatorics also raises several auxiliary questions which sometimes go beyond the area of concurrency.

In particular, the starting point of our approach is always to describe how to *decompose* the objects under study into smaller objects. In Chapter 1, this is expressed as a combinatorial specification, which is one the most powerful ways to achieve this since it automatically gives access to several results from [FS09]. In Chapter 2, this takes the form of a bijection describing more explicitly how to "break" the objects into smaller pieces. In the second case in particular, some of the smaller pieces are elementary and rather common combinatorial objects, such as combinations or variations. Incidentally, algorithms for sampling or unranking these objects are building blocks, not only for our use cases, but also in many other contexts so that studying how to implement them efficiently is a central question.

We dedicate the last chapter of this thesis to implementing and enhancing common algorithms or general-purpose algorithms which are used in the rest of our work, but also have a broader impact. Our main concern is set on the efficiency of the algorithms, especially when it comes to sampling primitive combinatorial objects. Our contributions in this direction are the development of a linear uniform sampler of variations and the improvement of classical algorithms to unrank combinations. In particular, the techniques used to improve the existing algorithms are not specific to the objects considered here and can be easily adapted to other algorithms. We also dive into the practical implementation of *Boltzmann* samplers which have been used for some experiments of Chapter 1. Boltzmann sampling is a general-purpose framework for obtaining approximate-size efficient random samplers from combinatorial specifications. Although the technique is not new and has a rich literature, the details of their implementation are usually not covered. One of our contributions is to describe precisely a generic Boltzmann sampler implementation as well as new optimisations. Most importantly, we describe a technique to let the users of a Boltzmann sampling library specify how to build the generated objects so as to make such a library more easy to integrate into an existing code-base.

Notations and generalities

We present here briefly a restricted list of notations and results that are used throughout the thesis.

Multiplication function

Many algorithms in this thesis involve arithmetic computations over unbounded integers. The bit-complexity of these algorithms, that is their complexity in terms of bitwise operations, of the different arithmetic operations over unbounded integers vary significantly from one algorithm to the other. Even the theoretical optimal bit-complexity of the multiplication is still an open problem. For these reasons, it is convenient to express some complexity bounds using a "multiplication function" $\mathcal{M}(n)$ which expresses the bit-complexity of the multiplication of two integers with at most n bits. We assume three properties of this function [Bos+17].

1. It is at least linear $n \leq \mathcal{M}(n)$.
2. It is super-additive $\mathcal{M}(n_1) + \mathcal{M}(n_2) \leq \mathcal{M}(n_1 + n_2)$.
3. And finally it is at most quadratic $\mathcal{M}(n) = O(n^2)$, that is the algorithm at use it at least as good as the naive multiplication algorithm.

In fact, the bit-complexity $\mathcal{M}(n)$ of the multiplication is conjectured to be of order $O(n \ln n)$ by Schönhage and Strassen in [SS71], but in practice a variety of algorithms are used (depending on the value of n) ranging from the naive "text-book" algorithm (of complexity $O(n^2)$) to the Schönhage–Strassen algorithm (of complexity $O(n \ln n \ln \ln n)$).

A particular case of interest of integer multiplication is when one of the operands is significantly larger than the other. In this case, we use the following upper-bound for the cost of the multiplication.

Lemma 1. *The bit-complexity of the multiplication of two integers with respectively n_1 and n_2 bits with $n_1 < n_2$ is bounded by*

$$\left\lceil \frac{n_2}{n_1} \right\rceil \mathcal{M}(n_1) + O(n_2).$$

Proof. This bound is obtained by decomposing the second number in base 2^{n_1} and in performing the multiplication in this base. This incurs $\lceil n_2/n_1 \rceil$ multiplications of numbers with at most n_1 bits and the decomposition and carry propagation incurs an extra linear term. □

Basic probability distributions

The random sampling algorithms presented in this thesis will make use of a few basic primitives for generating integers and Boolean variables.

The Unif function takes an integer interval of the form $[\![n; m]\!]$ as an argument and returns a uniform integer from this interval. This function consumes $O(\log(m-n))$ random bits in average.

The Bern function takes a real number $p \in [0; 1]$ as an argument and returns true with probability p and false with probability $(1 - p)$. If the (potentially infinite) binary representation of p is known, or if p is given in the form of a fraction, generating a Bernoulli variable of parameter p costs $O(1)$ random bits in average.

The Geom function takes a real number $p \in [0; 1]$ as an argument and generates integers following the geometric distribution, that is $\mathbb{P}[\textsc{Geom}(p) = k] = (1 - p)^k p$. Similarly, the Pois function takes a real number $\lambda > 0$ as an argument and generates integers following the Poisson distribution of parameter λ, that is $\mathbb{P}[\textsc{Pois}(\lambda) = k] = \frac{\lambda^k e^{-\lambda}}{k!}$. Both functions can be implemented using Algorithm 1, by using a different sequence p_k in each case. The values of p_k should be computed on the fly.

Algorithm 1 Common algorithm of the generation of geometric and Poisson variables

```
r ⟵ Uniform real number from [0; 1]
k ⟵ 0
while r ≥ p_k do
    r ⟵ r − p_k
    k ⟵ k + 1
return k
```

Geometric distribution:
$p_0 = p \quad \text{and} \quad p_{k+1} = (1 - p)p_k$

Poisson distribution:
$p_0 = e^{-\lambda} \quad \text{and} \quad p_{k+1} = \frac{\lambda p_k}{k+1}$

On-line Encyclopedia of Integer Sequences

In some places, we will refer to sequences stored in the On-line Encyclopedia of Integer Sequences (`https://oeis.org`) or OEIS for short. Such a sequence will be referred to using its identifier in the OEIS with is of the form `AXXXXXX`, for instance: `A007526`.

Chapter 1

Statistical analysis of non-deterministic fork-join programs

This chapter is devoted to the first facet of this thesis: the analysis of the state-space of concurrent programs via random generation. The approach presented here as well as some of the results have been published in the paper [GPP20][1].

The outline of this chapter is as follows. In Section 1.2, we present a first version of the program class of non-deterministic fork-join programs. We introduce the notion of global choice, which characterises the influence of non-determinism in this class and we provide precise quantitative results on this notion. We then describe a counting algorithm and a uniform random sampler of executions for such programs. In Section 1.3, we extend the model with loops and offer an alternative approach to random generation. Finally, in Section 1.4, we study the execution prefixes of programs with loops, both from a quantitative and an algorithmic point of view. At the end of each section, we support all the proposed algorithms with an experimental evaluation of their performance, thus establishing the tractability of our approach.

1.1 Context and related work

Analysing the state-space of concurrent programs is a notoriously difficult task, if only because of the infamous *state explosion* problem. Several techniques have been developed to mitigate this explosion: symbolic encoding of the state-space, partial order reductions, exploiting symmetries, etc.

An alternative approach is to adopt a probabilistic point of view, by developing statistical analysis techniques such as [Den+12]. The basic idea is to generate random executions from program descriptions, sacrificing exhaustiveness for the sake of tractability. This idea of empowering formal verification with probabilistic tools has been first presented in [GS05] where the authors introduce the notion of Monte-Carlo model-checking. An important question raised in [GS05] is: how to control the distribution of the sampled objects? There is indeed a crucial difference between generating an *arbitrary* execution and generating a *controlled* random

[1] [GPP20] "Statistical Analysis of Non-Deterministic Fork-Join Processes" has been published in the proceedings of the ICTAC conference in 2020. A longer version of this paper has been submitted to the TCS journal this year.

execution according to a known (typically the uniform) distribution. Only the latter allows to estimate the coverage of the state-space of a given analysis. Moreover, the *uniform* distribution plays an important role for that matter as it *a priori* gives the best coverage in the absence of any further information. Later the authors of [Oud+11] developed a uniform random sampler of *lassos*, which are linked to the verification of temporal-logic properties over potentially *infinite* executions, thus enabling *uniform* Monte-Carlo model-checking. However, their approach relies on the explicit, costly construction of the whole state-space, hence making it impractical except for small processes.

Our study combines viewpoints and techniques from concurrency theory and combinatorics. A similar line of work exists for the so-called "true concurrency" model (by opposition to the interleaving semantics that we use in our study) based on the trace monoid using *heaps combinatorics* (see [KMM02; AM15]). To our knowledge these only address the parallelism issue and not non-determinism *per se*. In [BMS17], the authors cover the problem of the uniform random generation of words in a class of synchronised automata. This approach is able to cover a slightly more expressive set of programs but this comes at the cost of the construction of a product (synchronizing) automaton of exponential size in the worst case. Finally [DPRS10] studies the random generation of executions in a model similar to the one we cover by extending the framework of Boltzmann sampling. Although Boltzmann samplers are usually fast, they turn out to be impractical in this context because of the heavy symbolic computations imposed by the interplay between parallelism and synchronisation.

The approach developed in this thesis builds on top of previous work on this topic where different operators of concurrency have been studied independently. Pure parallelism, in the absence of non-determinism and without synchronisation was analysed thoroughly in [BGP12] and [BGP16]. These two articles give precise quantitative results on the typical shape of the process behaviours and the average number of executions of the modelled programs. The question of the uniform random generation of executions is treated using a fast, ad hoc algorithm exploiting a *hook-length* formula counting the number of executions of a given process. Non-determinism, first discussed in [BGP13], is a simple model with only parallelism and without any form of synchronisation either. The article introduces algorithmic tooling for encoding the state-space in polynomial space and gives first quantitative results on the expressiveness of the non-deterministic choice operator. Whereas we could use similar tools to tackle the loop-free fragment of our language, we show in this chapter that this approach hits a wall when introducing loops and requires a new encoding of the state-space for the non-determinism to be studied in a more expressive language. A form of synchronisation similar to the one we have here has been studied in [BDGP17a] and [Die17], though in the absence of non-determinism. Our approach in Section 1.2 of the present Chapter consists in untangling the non-determinism from the other features of the language so that part of the sampling process can be delegated to the algorithm developed in [BDGP17a]. Once again, this process does not scale to the introduction of loops in the language in Section 1.3 and we need to develop a different approach.

1.2 A combinatorial interpretation of non-deterministic fork-join programs

The goal of this section is to introduce the combinatorial tools that will be used throughout the chapter and to study a first class of concurrent programs featuring a fork-join programming style with non-determinism. The interest of this class is that it showcases how different features of concurrency such as the interleaving semantics of the pure merge operator [BGP16], series-parallel synchronisation [BDGP17a] and non-determinism [BGP13] can be integrated and studied in a unified framework.

In Section 1.3, this class will be extended with a loop construction in order to gain more expressiveness. This extension has several technical implications on the combinatorial framework at use here, which will be discussed.

1.2.1 Non-deterministic Fork-Join processes (without loops)

Definition 1 (Non-deterministic fork-join programs)**.** *Given a set of symbols $\mathcal{A}$ representing the "atomic actions" of the language, we define the class of non-deterministic fork-join programs (over this set $\mathcal{A}$), denoted NFJ as follows:*

$$\begin{array}{lcll} P, Q & ::= & P \parallel Q & \textit{parallel composition} \\ & \mid & P; Q & \textit{sequential composition} \\ & \mid & P + Q & \textit{non-deterministic choice} \\ & \mid & a \in \mathcal{A} & \textit{atomic action} \end{array}$$

It is important to mention now that we will not specify further what the content of the atomic actions is. We treat them as black boxes and assume all action names within a term to be different. We also consider programs up to injective relabelling, so that $(a \parallel b)$ and $(d \parallel c)$ represent the same program. Our focus is set on the order in which these actions can be fired and scheduled by the different operators of the language. In other words, we study the *control-flow* of concurrent programs as an approximation of their behaviour. In all our examples we use lower-case Roman letters as unique identifiers to distinguish between actions.

We give NFJ an *interleaving* semantics, which means that an execution is seen as a *sequence* of small atomic steps and that the executions of $P \parallel Q$ are all the possible interleavings of an execution of P and an execution of Q. We start by defining a "step" relation of the form $P \xrightarrow{a} P'$ between two programs and an atomic action describing one small computation step. When we have $P \xrightarrow{a} P'$, we say that "program P reduces to P' by firing a". The inference rules defining the step relation are given in Figure 1.1 on the following page.

As a convenience, we allow the program on the right-hand-side of the step relation to be either a regular NFJ program or a special symbol 0 representing a program that has completed its execution. We also consider the following rewriting rules allowing to use 0 with the parallel and sequential composition operators in the conclusions of rules (Lpar), (Rpar) and (seq):

$$(0; P) = P$$
$$(P \parallel 0) = (0 \parallel P) = P$$

$$\frac{}{a \xrightarrow{a} 0}\ (\text{act}) \qquad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}\ (\text{Lpar}) \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}\ (\text{Rpar})$$

$$\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q}\ (\text{seq}) \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\ (\text{Lchoice}) \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\ (\text{Rchoice})$$

Figure 1.1: Semantic of NFJ given as a set of inference rules defining a step relation

We are now equipped to define the executions of the language as a sequence of steps ending on the empty program 0.

Definition 2 (Execution). *An execution of an NFJ program P_0 is a sequence of steps of the form*

$$P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \ldots \xrightarrow{a_{n-1}} P_{n-1} \xrightarrow{a_n} P_n = 0$$

where for all i, the step $P_{i-1} \xrightarrow{a} P_i$ is a proof-tree*, that is it contains all the applied rules and not simply its conclusion. We refer to the set of all possible executions of a program as its* state-space*.*

In the absence of ambiguity, which is the case in this section, a proof-tree can be safely identified with its conclusion. As an example, an execution of the program $P = (a; b) \parallel (c + d)$ may fire either a, c or d at the first step. The proof-trees corresponding to these three possible first steps are given below.

$$\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} 0}\ (\text{act})}{a; b \xrightarrow{a} b}\ (\text{seq})}{(a; b) \parallel (c + d) \xrightarrow{a} b \parallel (c + d)}\ (\text{Lpar}) \qquad \dfrac{\dfrac{\dfrac{}{c \xrightarrow{c} 0}\ (\text{act})}{c + d \xrightarrow{c} 0}\ (\text{Lchoice})}{(a; b) \parallel (c + d) \xrightarrow{c} a; b}\ (\text{Rpar})$$

$$\dfrac{\dfrac{\dfrac{}{d \xrightarrow{d} 0}\ (\text{act})}{c + d \xrightarrow{d} 0}\ (\text{Rchoice})}{(a; b) \parallel (c + d) \xrightarrow{d} a; b}\ (\text{Rpar})$$

Definition 2 gives a natural semantics to NFJ. However it does not highlight the influence of non-determinism in the structure of the state-space. It is interesting to separate the application of the choices from the interleaving semantics of the fork-join core of the language and, to this end, we use the notion of *global choice*. Informally, a global choice of a program P is a program obtained by selecting one of the two alternatives in each sub-term of the form $P_1 + P_2$ and removing the other.

Definition 3 (Global choices)**.** *The set of global choices of P, denoted* choices(P) *is a set of* choice-free *programs obtained from P inductively as follows:*

$$\begin{aligned}
\text{choices}(a) &= \{a\} \\
\text{choices}(P+Q) &= \text{choices}(P) \cup \text{choices}(Q) \\
\text{choices}(P \parallel Q) &= \left\{ (P' \parallel Q') \mid P' \in \text{choices}(P); Q' \in \text{choices}(Q) \right\} \\
\text{choices}(P;Q) &= \left\{ (P';Q') \mid P' \in \text{choices}(P); Q' \in \text{choices}(Q) \right\}
\end{aligned}$$

With this notion at hand, the non-determinism can be untangled from the interleaving semantics since an execution can now be seen as the combination of a global choice and an execution of this global choice. The selection of the global choice carries all the non-determinism coming from the choices of the program whereas the execution of this global choice contains all the expressiveness of the interleaving semantics. Moreover, it is easy to prove that an execution of a choice-free program fires *every* atomic action in the program exactly once. Therefore a global choice containing n atomic actions only has executions of length n and such an execution can be identified to a labelling of its actions with integers from the interval $[\![1;n]\!]$ corresponding to their position in the sequence of steps.

For instance, the program $P = m; (w \parallel ((t + (c;g)); (s+n); p)); e$ models the beverage vending machine pictured in Figure 1.2 on the next page. One of its possible executions corresponds to firing the following sequence of actions (in this order) m, t, n, w, p, e and the global choice corresponding to this execution is $P' = m; (w \parallel (t;n;p)); e$. Figure 1.2 on the following page gives a graphical representation of this execution as a labelling of P'. In the rest of this section, we will rely on this second point of view on executions to reason about them. To this end, we now introduce the combinatorial tools that we will need to model both the class of NFJ programs as a whole, and the set of executions of a given program, as combinatorial objects. This will enable us to analyse them using tools from analytic combinatorics.

1.2.2 The combinatorial toolset part 1: modelling programs

We introduce here the notions of combinatorial class, combinatorial specification, and generating function. This is a brief introduction as we limit ourselves to the tools that are necessary to tackle the problems covered in the present thesis. An in depth presentation of the techniques used here can be found in [FS09].

Definition 4 (Combinatorial class)**.** *A combinatorial class C is a set of objects equipped with a size function $|\cdot| : C \longrightarrow \mathbb{N}$ such that for all $n \in \mathbb{N}$ the set $C_n = \{c \in C \mid |c| = n\}$ of objects of size n of C is finite.*

For example, one combinatorial class of interest for us is the class of all NFJ programs where the size $|P|$ of a program P is defined as the number of atomic actions it contains, that is:

$$|(P \parallel Q)| = |(P+Q)| = |(P;Q)| = |P| + |Q|$$
$$|a| = 1$$

Table 1.1 on the next page gives the list of all NFJ programs of size at most 3 and their number.

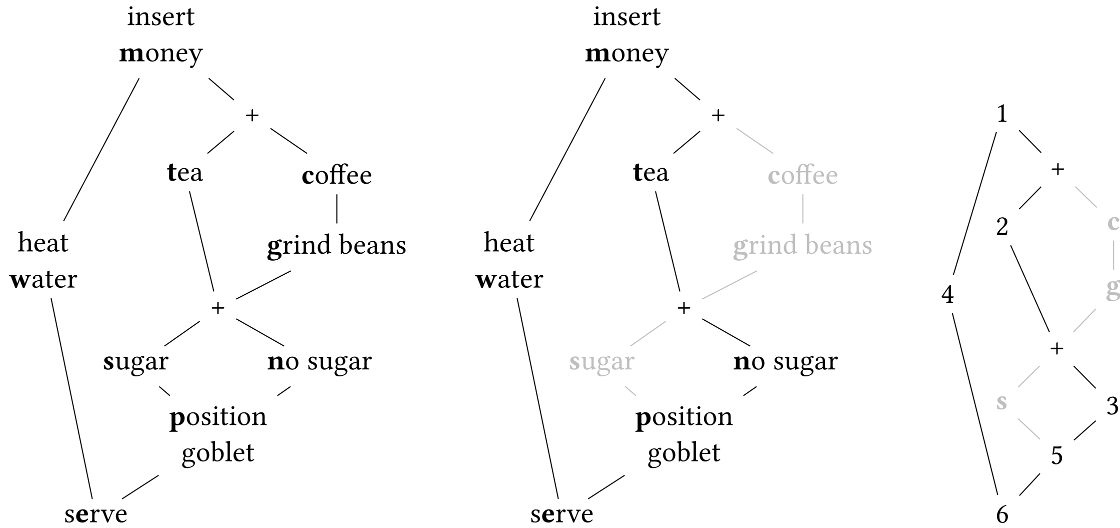


Figure 1.2: A simple beverage vending machine modelled by $P = m; (w \parallel ((t + (c; g)); (s + n); p)); e$, one of its 4 global choices and one of the 4 possible labellings of this global choice, representing the firing sequence m, t, n, w, p, e. This program has 18 executions in total.

Table 1.1: All NFJ programs of size at most 3 and their number

n	all programs of size n	number
1	a	1
2	$(a + b), (a \parallel b), (a; b)$	3
3	$(a + (b + c)), (a + (b \parallel c)), (a + (b; c)), ((a + b) + c), ((a \parallel b) + c), ((a; b) + c)$ $(a \parallel (b + c)), (a \parallel (b \parallel c)), (a \parallel (b; c)), ((a + b) \parallel c), ((a \parallel b) \parallel c), ((a; b) \parallel c)$ $(a; (b + c)), (a; (b \parallel c)), (a; (b; c)), ((a + b); c), ((a \parallel b); c), ((a; b); c)$	18

Since the number of elements of a given size of a combinatorial class C is finite, it is possible to define its generating function $C(z)$ as the formal power series $C(z) = \sum_{n \geq 0} c_n z^n$ where c_n is the cardinality of C_n, that is the number of objects of size n in C. The reason behind considering a generating function rather than working on the sequence c_n directly is twofold. First, generating functions behave nicely with respect to high-level operations on combinatorial classes, like the Cartesian product or the disjoint union. This often allows to obtain recurrence relations on the sequences — and even sometimes an explicit formula — in a systematic and elegant way with few manual computations. Second, this enables the use of analytic techniques when this function converges, allowing to obtain information on the asymptotic behaviour of the sequences via complex analysis. This approach has been used successfully in a variety of contexts and has been popularised by the book [FS09].

As we just mentioned, generating functions behave nicely with respect to some high-level operations on combinatorial classes. For instance we define the disjoint union C of two classes $\mathcal{A}$ and $\mathcal{B}$, denoted by $C = \mathcal{A} + \mathcal{B}$, as the combinatorial class whose underlying set is the disjoint

union of the elements of $\mathcal{A}$ and the elements of $\mathcal{B}$ and whose size function $|\cdot|_{\mathcal{C}}$ is such that

$$|c|_{\mathcal{C}} = \begin{cases} |c|_{\mathcal{A}} & \text{if } c \in \mathcal{A} \\ |c|_{\mathcal{B}} & \text{if } c \in \mathcal{B} \end{cases} \quad \text{where} \quad \begin{array}{l} |\cdot|_{\mathcal{A}} \text{ is the size function of } \mathcal{A}; \\ |\cdot|_{\mathcal{B}} \text{ is the size function of } \mathcal{B}. \end{array}$$

It is easy to check that the generating function $C(z)$ of the class $\mathcal{C} = \mathcal{A} + \mathcal{B}$ satisfies the formula $C(z) = A(z) + B(z)$ where $A(z)$ and $B(z)$ denote the generating functions of $\mathcal{A}$ and $\mathcal{B}$. Using similar notations, we can also define the Cartesian product $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ of two combinatorial classes $\mathcal{A}$ and $\mathcal{B}$ where the size of an element (a, b) of $\mathcal{C}$ is $|(a, b)|_{\mathcal{C}} = |a|_{\mathcal{A}} + |b|_{\mathcal{B}}$. Hence, the set $\mathcal{C}_n$ of objects of $\mathcal{C}$ of size n is $\left\{(a, b) \mid a \in \mathcal{A}_i, b \in \mathcal{B}_j, i + j = n\right\}$. As a consequence the cardinality c_n of $\mathcal{C}_n$ is $\sum_{i+j=n} a_i b_j$ where a_i is the cardinality of $\mathcal{A}_i$ and b_j is the cardinality of $\mathcal{B}_j$. Thus, we obtain a simple expression for the generating function of $\mathcal{C}$, that is $C(z) = A(z) \cdot B(z)$.

It follows that if one is able to describe a combinatorial class using only these constructions (and possibly recursion), then it is straightforward to obtain an equation satisfied by the generating function of the class. Such a description is called a *combinatorial specification*. The process of finding a specification for the system under study and then applying automatic rules to derive its generating function is called the *symbolic method*. Table 1.2 gives a few of the constructions we will need in this section and their translations in terms of generating functions. The neutral class $\mathcal{E}$ is not useful for this section but will be later in the next section.

Table 1.2: Some constructions of the symbolic method

	$\mathcal{C}$	$C(z)$
Neutral class: one element of size 0	$\mathcal{E}$	1
Atomic class: one element of size 1	$\mathcal{Z}$	z
Disjoint union	$\mathcal{A} + \mathcal{B}$	$A(z) + B(z)$
Cartesian product	$\mathcal{A} \times \mathcal{B}$	$A(z)B(z)$

As an example of application, it is not too difficult to obtain a specification of the class $\mathcal{F}$ of NFJ programs:

$$\mathcal{F} = \mathcal{Z} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F}. \tag{1.1}$$

This specification makes use of the disjoint union, the Cartesian product and a new construction $\mathcal{Z}$ called the *atomic* class. This is the class containing only one element of size one. Here it represents the atomic actions of the language, indeed the program made of only one action has size one and there is only one such program (up to relabelling). The other three terms represent the three other constructions of the language. A program that is not reduced to a single action is either a parallel composition, a sequential composition or a choice between two programs. Moreover these three sets are disjoint, hence the disjoint union. Also, note that the equal sign here denotes an isomorphism rather than an equality, in the sense that there is a bijection between the two sides of the equality that preserves the size. The equal sign, when used in a specification, will always denote an isomorphism rather than a strict equality.

From equation (1.1) and using the transformation rules of the symbolic method recalled in Table 1.2, we obtain that the generating function f of NFJ programs satisfies $f(z) = z + 3f(z)^2$.

This equation can be solved explicitly so that we have a closed formula for f:

$$f(z) = \frac{1 - \sqrt{1 - 12z}}{6}. \tag{1.2}$$

This leads us to our first result on the number of NFJ programs.

Theorem 1 (Number of NFJ programs). *For $n > 0$, the number f_n of NFJ programs containing exactly n atomic actions is given by $f_n = \frac{3^n}{12n-6}\binom{2n}{n}$. Moreover we have $f_n \underset{n\to\infty}{\sim} \frac{12^{n-1}}{\sqrt{n^3\pi}}$.*

Proof. Recall that the number f_n of programs of size n is the coefficient of degree n in the generating function $f(z)$, which we denote by $[z^n]f(z)$. Moreover we have that

$$\begin{aligned} \sqrt{1-u} &= -\sum_{n\geq 0} \frac{4^{-n}}{2n-1}\binom{2n}{n} u^n \\ \text{hence} \quad 1 - \sqrt{1-12z} &= \sum_{n\geq 1} \frac{3^n}{2n-1}\binom{2n}{n} z^n \\ \text{and therefore} \quad f_n &= \frac{3^n}{12n-6}\binom{2n}{n}. \end{aligned}$$

The equivalent for f_n is then obtained by applying Stirling's formula. □

The number of NFJ programs is not interesting as such, but we give it here for two reasons. First it gives an example of application of the symbolic method which demonstrates how quickly one can get to a counting formula, and an asymptotic estimation, using this approach. And second, we need this result as an auxiliary result to prove a more insightful quantitative theorem on the class of NFJ programs: an NFJ program has, in average, relatively few global choices.

The natural intuition is that, since a global choice can be any combination of local choices, the typical number of global choices of a program should grow exponentially with the size of the program. This intuition is correct, but what is less natural is that the growth rate of the number of global choices is quite small. In Theorem 2 we prove that the typical number of global choices, that is their average number over all programs of size n, grows as fast as about 1.11438^n.

Theorem 2 (Average number of global choices). *The average number of global choices of an NFJ program of size n is equivalent $A\cdot B^n$ when $n \longrightarrow \infty$ for some constants A and B such that $A \approx 6.89446$ and $B = \frac{49}{27+12\sqrt{2}} \approx 1.11438$.*

The practical implication of this somewhat surprising result is that, for an average program of reasonably small size, it might be possible to enumerate *all* global choices because their number is such a "small" exponential. As an example, the average number of global choices for programs of size $n = 100$ is approximatively 348 261.

Proof of Theorem 2. The idea to prove Theorem 2 is to count programs annotated with one of their global choices. In other words, we consider the combinatorial class $\mathcal{G}$ of all the pairs of the form (P, P') where $P' \in \text{choices}(P)$ and where the size function of $\mathcal{G}$ is defined by $|(P, P')| = |P|$.

The number of objects of size n in $\mathcal{G}$ is therefore the sum, over all programs of size n, of their number of global choices. In order to get the *average* number of global choices given in the statement of the theorem, it will suffice to divide this quantity by the number of programs of size n obtained in Theorem 1.

The class $\mathcal{G}$ can be specified as follows:

$$\mathcal{G} = \mathcal{Z} + \mathcal{G} \times \mathcal{G} + \mathcal{G} \times \mathcal{G} + (\mathcal{G} \times \mathcal{F} + \mathcal{F} \times \mathcal{G}). \tag{1.3}$$

And the combinatorial interpretation of the above formula is the following:

- A single action has only one global choice. Hence, there is only one annotated program (P, P') in $\mathcal{G}$ where $P = a$. This is specified by $\mathcal{Z}$.
- A global choice of $P \parallel Q$ is by definition the parallel composition of a global choice of P and a global choice of Q. Hence, the set of annotated programs where the outermost constructor is $\parallel$ is isomorphic to a Cartesian product of $\mathcal{G}$ with itself.
- The same reasoning applies to the sequential composition.
- Finally, the most interesting case is that of the choice construction. A global choice of a program of the form $P + Q$ is either a global choice of P or a global choice of Q. Hence, the set of pairs of the form $(P + Q, R)$ in $\mathcal{G}$ are such that $(P, R) \in \mathcal{G}$ (or $(Q, R) \in \mathcal{G}$) and the second program Q (or P) is a regular, non-annotated, NFJ program. Therefore the sub-class of such terms is isomorphic to $\mathcal{G} \times \mathcal{F} + \mathcal{F} \times \mathcal{G}$.

From (1.3) we obtain that the generating function $g(z)$ of annotated programs satisfies the following equation where f is the generating function of regular (non-annotated) NFJ program:

$$g(z) = z + 2g(z)^2 + 2g(z)f(z). \tag{1.4}$$

Again, this equation can be solved explicitly which yields $g(z) = \frac{1-2f(z)-\sqrt{\Delta(z)}}{4}$ where Δ is given by $\Delta(z) = (1 - 2f(z))^2 - 8z$ expands to $\frac{1}{9}\left(5 - 84z + 4\sqrt{1 - 12z}\right)$. Obtaining an explicit formula for the number g_n of annotated programs would require to extract the coefficient of degree n in the power series expansion of $g(z)$, which would be extremely tedious.

Instead we resort to singularity analysis. The function Δ is well-defined and decreasing in the interval $[0; \frac{1}{12}]$ and $\Delta(\frac{1}{12}) = -\frac{2}{9} < 0$ so there exists a unique ρ_g in this interval such that $\Delta(\rho_g) = 0$. Let $u = \sqrt{1 - 12\rho_g}$, we have that $9\Delta(\rho_g) = 5 - 7(1 - u^2) + 4u = 0$, which we can solve explicitly and yields $u = \frac{3\sqrt{2}-2}{7}$ and therefore $12\rho_g = \frac{27+12\sqrt{2}}{49} < 1$. This allows to write:

$$g(z) = \frac{1 - 2f(z)}{4} - h(z)\sqrt{1 - \frac{z}{\rho_g}}$$

$$\text{where} \quad h(z) = \sqrt{\Delta(z)\left(1 - \frac{z}{\rho_g}\right)^{-1}} \quad \text{is analytic in } \left\{ z \in \mathbb{C} \mid |z| < \frac{1}{12} \right\}$$

Therefore the transfer theorem from [FS09, Thm. VI.3 p. 390] applies to $g(z)$ and its n-th coefficient g_n satisfies $g_n \sim \frac{h(\rho_g)}{2\sqrt{\pi}} n^{-\frac{3}{2}} \rho_g^{-n}$. Furthermore, we have that $h(\rho_g) = \sqrt{-\rho_g \Delta'(\rho_g)}$ which can be computed explicitly. This allows us to conclude the proof since the average number of global choices over programs of size n is $g_n/f_n \sim 6h(\rho_g) \cdot (12\rho_g)^n$. □

The impact of symmetries

A natural question to ask, regarding Theorem 2 and its proof, is "how would commutativity and associativity affect this result?". We can indeed see in Table 1.1 on page 13 that even for small sizes, many programs can be obtained from one another by flipping the operands of the parallel or the choice operator for instance, and it is clear from the semantics of NFJ that doing so does not change the state-space of programs. It is actually possible to answer this question using similar tools as before, although the proofs become significantly more technical. In this sub-section we state the analogue of Theorem 2 in a refined model which takes the aforementioned symmetries into account.

We now consider all three constructions of the language to be associative so that for instance $(a \parallel (b \parallel c)) = ((a \parallel b) \parallel c)$, which we will now write $(a \parallel b \parallel c)$. Moreover we consider the parallel composition operator and the choice operator to be commutative so that $(a \parallel (b; c)) = ((b; c) \parallel a)$ for instance. This has the consequence that there are fewer NFJ programs of given size in this model than in the simpler one, and that programs with many symmetries will be given less importance when computing the *average* number of global choices. We establish the following result.

Theorem 3. *The average number of global choices for programs with n atomic actions, taken up to commutativity and associativity is equivalent to $A \cdot B^n$ when $n \longrightarrow \infty$ where $B \approx 1.11275$.*

It is extremely common in analytic combinatorics that going from *ordered* to *unordered* collections (in our context: from non-commutativity to commutativity) does not change the form the result but only the constants appearing in it. So the fact that the average number of global choices in this new model has the same behaviour is not surprising. However it is interesting that the new growth rate of the number of global choices is this close to the one we obtained in Theorem 2, and that it is slightly smaller. It is also possible to approximate the constant A but is extremely tedious and of little interest.

Taking these symmetries into account in the counting process requires to write new specifications for the class $\mathcal{F}$ of NFJ programs and for the class $\mathcal{G}$ of annotated NFJ programs, that is the set of pairs (P, P') where $P \in \mathcal{F}$ and P' is a global choice of P. In these specifications, associativity is expressed using n-ary rather than binary operators whereas commutativity requires to introduce two operators that we have not seen so far, the multi-set operator and an unusual "replication" operator.

Specification of the set of program Let $\mathcal{F}$ denote the class of NFJ programs, taken up to associativity and commutativity, and let $\mathcal{F}_;$, $\mathcal{F}_\parallel$ and $\mathcal{F}_+$ denote the sub-classes of $\mathcal{F}$ containing programs whose outermost constructors are respectively a sequence, a parallel composition and a choice.

The class $\mathcal{F}_;$ for instance contains all programs of the form $(P_1; P_2)$ for any two programs P_1 and P_2. But there is an ambiguity issue with this representation when one of P_1 or P_2 is itself of the form $(P_3; P_4)$, because there are at least two possible ways to represent the program. To circumvent this issue, we "flatten" nested sequence operators and say that an element of $\mathcal{F}_;$ is of the form $P_1; P_2; \dots; P_k$ where $k \geq 2$ and for all $1 \leq i \leq k$ we have $P_i \notin \mathcal{F}_;$. This yields the

specification $\mathcal{F}_; = \textsc{Seq}_{\geq 2}(\mathcal{F} \setminus \mathcal{F}_;)$ or, equivalently, $\mathcal{F}_; = \textsc{Seq}_{\geq 2}(\mathcal{Z} + \mathcal{F}_\| + \mathcal{F}_+)$, where $\textsc{Seq}_{\geq 2}(\cdot)$ denotes a sequence of at least 2 elements and is formally defined below.

Definition 5. *Given a combinatorial class $\mathcal{A}$ with no element of size 0, we define the class of sequences of elements of $\mathcal{A}$ as:*

$$\textsc{Seq}(\mathcal{A}) = \bigcup_{j \geq 0} \mathcal{A} \times \mathcal{A} \times \cdots \times \mathcal{A} \quad (j \text{ times})$$

For $k \geq 0$, we denote by $\textsc{Seq}_{\geq k}(\mathcal{A})$, the restriction of the above union to $j \geq k$.

Proposition 1. *Let $\mathcal{A}$ be a combinatorial class with no element of size 0 and let $A(z)$ denote its generating function. The generating function of $\textsc{Seq}_{\geq k}(\mathcal{A})$ is given by*

$$\frac{A(z)^k}{1 - A(z)}.$$

For $\mathcal{F}_+$ and $\mathcal{F}_\|$ we handle associativity the same way, but this does not solve the commutativity issue. The idea to express commutativity is to see the operands of a choice (resp. parallel composition) as a *multi-set* of programs rather than a sequence since the order in which they are written does not matter. Note that we do need a multi-set and not a set since two programs that are equal (up to renaming of the actions) may be used as two branches of a choice or may be composed in parallel. In combinatorics multi-sets are specified using the $\textsc{MSet}(\cdot)$ operator, defined below, which allows to write $\mathcal{F}_+ = \textsc{MSet}_{\geq 2}(\mathcal{F} \setminus \mathcal{F}_+)$ for instance. The treatment of this kind of operators relies on what is known as *Pólya theory* and is covered by the book [PR12].

Definition 6. *Given a combinatorial class $\mathcal{A}$ with no element of size 0, we define the class of multi-sets of elements of $\mathcal{A}$, denoted by $\textsc{MSet}(\mathcal{A})$, as the quotient of $\textsc{Seq}(\mathcal{A})$ by the following equivalence relation:*

$$(a_1, a_2, \ldots, a_k) \sim (a'_1, a'_2, \ldots, a'_\ell) \iff (k = \ell) \wedge \exists \sigma \in \mathfrak{S}_k, \forall\, 1 \leq i \leq k, a_{\sigma(i)} = a'_i$$

where $\mathfrak{S}_k$ denotes the set of permutations of $[\![1; k]\!]$.

The size of an element of $\textsc{MSet}(\mathcal{A})$ is defined as the sum of the sizes of its components (this is independent of the ordering). Moreover, for $k \geq 0$, the class $\textsc{MSet}_{\geq k}(\mathcal{A})$ is defined as the sub-class of $\textsc{MSet}(\mathcal{A})$ whose elements have at least k components, that is $\textsc{MSet}_{\geq k}(\mathcal{A}) = \textsc{Seq}_{\geq k}(\mathcal{A}) / \sim$.

Proposition 2. *Let $\mathcal{A}$ be a combinatorial class with no element of size 0 and let $A(z)$ denote its generating function. The generating function of the multi-set $\textsc{MSet}(\mathcal{A})$ is given by*

$$\exp\left(\sum_{j \geq 1} \frac{A(z^j)}{j}\right).$$

Putting together the two constructions introduced above, we get a specification for NFJ expressing associativity and commutativity.

$$\begin{aligned} \mathcal{F} &= \mathcal{Z} + \mathcal{F}_; + \mathcal{F}_\| + \mathcal{F}_+ \\ \mathcal{F}_; &= \textsc{Seq}_{\geq 2}(\mathcal{F} \setminus \mathcal{F}_;) \\ \mathcal{F}_\| &= \textsc{MSet}_{\geq 2}(\mathcal{F} \setminus \mathcal{F}_\|) \\ \mathcal{F}_+ &= \textsc{MSet}_{\geq 2}(\mathcal{F} \setminus \mathcal{F}_+) \end{aligned} \tag{1.5}$$

From this specification, one can derive a system of equations on the generating functions of these four classes and the singularity analysis of these functions yields the following theorem.

Theorem 4. *The asymptotic number f_n of NFJ programs, up to associativity and commutativity, satisfies*

$$f_n \underset{n\to\infty}{=} \gamma n^{-\frac{3}{2}} \rho^{-n}(1 + O(n^{-1}))$$

where $\rho \approx 0.13793576712500258$ and $\gamma \approx 0.12607642812680533$.

The value of $\rho^{-1} \approx 7.25$ here has to be compared with the growth rate 12 which we obtained in Theorem 1. A heuristic argument which partly explains this value is that each *syntactic* program of size n is made of $n - 1$ binary operators and that about $\frac{2}{3}$ of them are choices or parallel composition. Moreover, it is unlikely that the two operands of a binary operator are isomorphic so each of these $\frac{2}{3}(n - 1)$ commutative operators induce a symmetry. In total this accounts for about $2^{\frac{2}{3}n}$ symmetries. In turns out that $12/2^{\frac{2}{3}} \approx 7.56$ which is somewhat close to ρ^{-1}, the rest of the difference correspond to the associativity and the rough approximations of this argument.

Proof of Theorem 4. We follow here a similar proof to that of Theorem 2. That is we explicit the behaviour of the generating function $f(z)$ of NFJ programs near its dominant singularity and we deduce the number of programs using the transfer theorem from [FS09, Thm. VI.3 p. 390]. There is a major difference from Section 1.2 here though, which is that we do not look for an explicit solution to the functional equations. Instead we use complex analysis to describe the behaviour of the solution near its dominant singularity which is actually enough to get precise asymptotic results.

The translation of specification (1.5) in terms of generating functions yields the following system by the symbolic method. Note that by symmetry, $\mathcal{F}_+$ and $\mathcal{F}_\parallel$ have the same number of elements of each size so their respective generating functions $f_\parallel$ and f_+ are equal and we only use the former.

$$\begin{aligned} f(z) &= z + f_;(z) + 2f_\parallel(z) \\ f_;(z) &= \frac{1}{1 - (f(z) - f_;(z))} - 1 - (f(z) - f_;(z)) \\ f_\parallel(z) &= \exp\left(\sum_{j\geq 1} \frac{f(z^j) - f_\parallel(z^j)}{j}\right) - 1 - (f(z) - f_\parallel(z)) \end{aligned}$$

To simplify this system, we first observe that $f_;$ can be expressed as a function of f only, more precisely $f_;(z) = f(z)^2(1 + f(z))^{-1}$. Hence $f_;$ has the same radius of convergence as f. For $f_\parallel$ however no such simplification is possible and we resort to a classical argument for this kind of operators. First note that f and $f_\parallel$ have the same radius of convergence ρ and that $\rho < 1$. This can be proved for instance by observing that $\mathcal{Z} \times \mathcal{F}_; \subset \mathcal{F}_\parallel$ and $(\mathcal{Z} + \mathcal{F}_\parallel)^2 \subset \mathcal{F}_\parallel$ and by enumerating the programs belonging to the subset of $\mathcal{F}$ described by $\mathcal{F}'_; = \mathcal{Z} \times \mathcal{F}'_;$ and $\mathcal{F}_\parallel = (\mathcal{Z} + \mathcal{F}_;)^2$. As a consequence, we have that for all $j > 1$, the radius of convergence of $f(z^j)$ (and thus $f_\parallel$) is at

least $\sqrt{\rho} > \rho$. Hence, the exponential in the above formula can be split in two:

$$f_{\|}(z) = e^{f(z)-f_{\|}(z)} e^{\zeta_1(z)} - 1 - (f(z) - f_{\|}(z))$$
$$\text{where} \quad \zeta_1(z) = \sum_{j\geq 2} \frac{f(z^j) - f_{\|}(z^j)}{j} \quad \text{is analytic in } \left\{ z \mid |z| < \sqrt{\rho} \right\}.$$

As a consequence, we get that $f_{\|}(z) = \ln \frac{1}{1+f(z)} + f(z) + \zeta_1(z)$ and finally $f(z) = \zeta_2(z) + \phi(f(z))$ where ϕ is the analytic function defined below and $\zeta_2(z) = z + 2\zeta_1(z)$. Note that ϕ has no terms of degree 0 and 1, this will be important in the following.

$$\phi(u) = \frac{u^2}{1+u} + 2\ln\frac{1}{1+u} + u = \sum_{n\geq 2} (-1)^n \left(1 + \frac{2}{n}\right) u^n.$$

In order to get the asymptotic behaviour of f near its singularity, we first study the functional equation $y(x) = x + \phi(y(x))$ which admits a unique analytic solution in a neighbourhood of 0. By unicity of the solution, we get that $f(z) = y(\zeta_2(z))$ and that the radius of convergence ρ of f is the unique $z > 0$ such that $\zeta_2(z)$ is equal to the radius of convergence of y.

We solve the equation $y = x + \phi(y)$ by applying the Theorem 2.19 from [Drm09] after a simple change of variable so that it fits the hypotheses of the Theorem. Note that we actually need the weaker version of the theorem exposed in Remark 2.20 from the same book because our function ϕ has negative coefficients in its expansion. By introducing $\tilde{y} = \frac{y}{x} - 1$ we can reformulate the equation as in:

$$\tilde{y} = \frac{1}{x}\phi(x(1+\tilde{y})) = F(x, \tilde{y}).$$

The hypotheses of the Remark 2.20 of [Drm09] apply to this equation. In particular we find that $x_0 = y_0 - \phi(y_0)$ and $\tilde{y}_0 = \frac{\phi(y_0)}{x_0}$ where $y_0 = \frac{\sqrt{3}-1}{2}$ is the unique solution of $\partial_{\tilde{y}} F(x, \tilde{y}) = 1$ on the positive real axis. Hence, by [Drm09] there exists a unique solution $\tilde{y}$ to this equation, it is analytic for $|x| < x_0$ and furthermore there exist two functions $\tilde{h}_1$ and $\tilde{h}_2$ which are analytic around $x = x_0$ and such that locally around $x = x_0$ we have:

$$\tilde{y}(x) = \tilde{h}_1(x) - \tilde{h}_2(x)\sqrt{1 - \frac{x}{x_0}}$$
$$\text{besides,} \quad \tilde{h}_1(x_0) = \tilde{y}_0 \quad \text{and} \quad \tilde{h}_2(x_0) = \sqrt{\frac{2x_0 \partial_x F(x_0, \tilde{y}_0)}{\partial_{\tilde{y}^2} F(x_0, \tilde{y}_0)}} = \sqrt{\frac{2(y_0\phi'(y_0) - \phi(y_0))}{x_0^2 \phi''(y_0)}}.$$

As a consequence f is analytic in $|\zeta_2(z)| < x_0$ and its dominant singularity ρ is the unique $z > 0$ such that $\zeta_2(z) = x_0$. Moreover, there exist two analytic functions h_1 and h_2 such that locally around $z = \rho$ we have

$$\begin{aligned} f(z) &= h_1(z) - h_2(z)\sqrt{1 - \frac{z}{\rho}} \\ &= y_0 - x_0\tilde{h}_2(x_0)\sqrt{\frac{\rho\zeta_2'(\rho)}{x_0}}\sqrt{1 - \frac{z}{\rho}} + O(\rho - x). \end{aligned}$$

Finally, the transfer theorem (see [FS09, Thm. VI.3 p. 390]) allows us to deduce the asymptotic behaviour of the sequence f_n, counting the number of NFJ programs of size n, when n tends to the infinity. □

Numerical estimation of the constants Some arguments exposed above are not constructive. In particular the constant ρ is defined as the solution of an equation involving $f(z^j)$ and $f_\parallel(z^j)$ for all $j \geq 2$, for which we have no expression. In addition, the value of γ depends on the value of ρ and $\zeta_2'(\rho)$, this function being implicitly defined. It is however possible to numerically evaluate these constants with extremely good precision using a method explained in [FS09, Section VII.5], which we detail in this section.

The method is actually based on a simple idea. One starts by computing the first terms of the expansion in power series of f and $f_\parallel$ up to some degree m, this yields two polynomials $f^{[m]}$ and $f_\parallel^{[m]}$ of degree m. Then one uses these two expansions to approximate the ζ_2 function by $\zeta_2^{[m]}(z) = z + 2\sum_{j=2}^{m} j^{-1}(f^{[m]}(z^j) - f_\parallel^{[m]}(z^j))$ and we get an approximation of ρ by solving numerically the equation $\zeta_2^{[m]}(z) = x_0 = \ln\left(\frac{2+\sqrt{3}}{2}\right) - \frac{3\sqrt{3}-5}{2}$. Note that the function $\zeta_2^{[m]}$ is increasing on the positive real axis so the equation $\zeta^{[m]}(z) = x_0$ can be numerically solved by dichotomy. Finally, we let m grow until the approximation stabilises. Since the solution ρ of $\zeta_2(z) = x_0$ lies inside the disc of convergence of ζ_2, the approximation of $\zeta_2(z)$ by $\zeta_2^{[m]}(z)$ converges quickly and the approximation of ρ is extremely precise even for small values of m.

There is one subtlety though in the computation of the first terms of the expansion of $f_\parallel$ and $f_;$. We did not give a formula for computing the first terms of $\textsc{MSet}(\mathcal{A})$ given the first terms of $\mathcal{A}$. The usual approach to obtain such a relation is to exploit the formula for the derivative of the generating function of $\textsc{MSet}(\mathcal{A})$ which has a more convenient expression. In our case, we want to the first terms of $f_\parallel(z) = \exp(f(z) - f_\parallel(z) + \zeta_1(z)) - 1 - (f(z) - f_\parallel(z))$ whose derivative is:

$$\begin{aligned} f_\parallel' &= (f' - f_\parallel' + \zeta_1') \exp(f - f_\parallel + \zeta_1) - (f' - f_\parallel') \\ &= (f' - f_\parallel' + \zeta_1')(f + 1) - (f' - f_\parallel') \\ &= (f' - f_\parallel' + \zeta_1') \cdot f + \zeta_1' \\ \text{with} \quad \zeta_1'(z) &= \sum_{j \geq 2} z^{j-1}(f'(z^j) - f_\parallel'(z^j)). \end{aligned}$$

One can then obtain a recurrence relation for the coefficient of degree n of $f_\parallel'$ by extracting the term of degree n from both sides of the last equality. Since $f(0) = 0$ and since the sum starts at $j = 2$ in ζ_1', only coefficients of f' and $f_\parallel'$ of degree less than n appear on the right-hand-side of the relation.

Getting a recurrence relation to compute the n-th term of $f_;$ is more straightforward as there is no exponential to deal with. One can for instance use the following formula (note that both operands of the product are null at 0):

$$f_;(z) = f(z)(f(z) - f_;(z)).$$

Using the approach described above, one quickly gets a good approximation of ρ and $\zeta_2'(\rho)$, this second value being necessary to compute γ. As a rough indication of the speed of convergence of this approximation scheme, with our implementation using double precision floats, the sequence $m \mapsto \rho^{[m]}$ is stationary after $m = 16$ as we hit the limit of representatble numbers.

Specification of the set of annotated program Like in the proof of Theorem 2, in order to count the average number of global choices of programs of size n, one first counts the number of pairs (P, P') where P is a program of size n and P' is one of its global choices. We call such a pair an *annotated* program and we define its size as the size of its first component P.

Let $\mathcal{G}$ denote the class of annotated programs of size n and let $\mathcal{G}_;$, $\mathcal{G}_+$ and $\mathcal{G}_\parallel$ denote the classes of annotated programs whose outermost operator is respectively a sequential composition, a choice or a parallel composition. The equations defining $\mathcal{G}$, $\mathcal{G}_;$ and $\mathcal{G}_\parallel$ are straightforward to obtain as they follow closely their non-annotated counterparts $\mathcal{F}$, $\mathcal{F}_;$ and $\mathcal{F}_\parallel$:

$$\begin{aligned} \mathcal{G} &= \mathcal{Z} + \mathcal{G}_; + \mathcal{G}_\parallel + \mathcal{G}_+ \\ \mathcal{G}_; &= \textsc{Seq}_{\geq 2}(\mathcal{G} \setminus \mathcal{G}_;) \\ \mathcal{G}_\parallel &= \textsc{MSet}_{\geq 2}(\mathcal{G} \setminus \mathcal{G}_\parallel). \end{aligned} \tag{1.6}$$

The case of the choice however, requires more work as we need to express the fact that one of the branches of the choice is executed and the others are not. In order to specify this, we reason on the executed branch of the choice and on the sub-set of the other branches which are isomorphic to it. The executed branch (P_0, P_0') belongs to $(\mathcal{G} \setminus \mathcal{G}_+)$ for the same reason the branches of regular (non-annotated) choice programs $\mathcal{F}_+$ belonged to $(\mathcal{F} \setminus \mathcal{F}_+)$ in the previous section. Moreover, among the other branches, $k \geq 0$ branches $P_1, P_2, \dots, P_k$ may be isomorphic to P_0, that is to say that all of the P_i are copies of P_0 with different atom names. Although the choice operator is associative in this section, we have to specify that the program $P = P_0 + P_1 + P_2 + \cdots + P_k$ has $(k+1)$ distinct choices (one of each P_i) according to our specification. One way to achieve this at the specification level, is to artificially *partition* the branches of P into two sets. We fix an arbitrary ordering of the P_i and we distinguish between the P_i that are "before" P_0 and those that are "after" P_0. Thus, an annotated choice program in $\mathcal{G}_+$ is composed of an annotated branch $(P_0, P_0') \in \mathcal{G} \setminus \mathcal{G}_+$, a possibly empty set of copies of P_0 considered to be before P_0, a possibly empty set of copies of P_0 considered to be after P_0 and a possibly empty multi-set of other branches (different from P_0). As a consequence $\mathcal{G}_+$ is specified by:

$$\bigcup_{(P_0,P_0') \in \mathcal{G} \setminus \mathcal{G}_+} \textsc{MSet}(\{P_0\}) \times \left\{ (P_0, P_0') \right\} \times \textsc{MSet}(\{P_0\}) \times \textsc{MSet}(\mathcal{F} \setminus \mathcal{F}_+ \setminus \{P_0\}) \setminus \left\{ (P_0, P_0') \right\}.$$

Note that the removal of the term $\left\{ (P_0, P_0') \right\}$ on the right captures the fact that a choice term cannot be reduced to one single branch. Also note that the term $\textsc{MSet}(\{P_0\}) \times \textsc{MSet}(\mathcal{F} \setminus \mathcal{F}_+ \setminus \{P_0\})$ can be simplified to $\textsc{MSet}(\mathcal{F} \setminus \mathcal{F}_+)$. This can be interpreted by saying that the copies of P_0 considered to be after P_0 can be grouped together with the multi-set of other branches (different from P_0) so that it forms one single multi-set of non-annotated programs. Furthermore, the remaining terms can be simplified too by observing that the j remaining copies of P_0 can be grouped together with (P_0, P_0') and that the $(j+1)$-tuples of the form $(P_0, P_0, \dots, P_0, (P_0, P_0')) \in \mathcal{F}^j \times \mathcal{G}$ have the same size as the $(j+1)$-tuples of the form $((P_0, P_0'), (P_0, P_0'), \dots, (P_0, P_0')) \in \mathcal{G}^{j+1}$ and are trivially in bijection with them. We introduce a "replication" operator which can specify this kind of tuples.

Definition 7. *Let $\mathcal{A}$ be a combinatorial class with no element of size 0, we define the class of replicas of elements of $\mathcal{A}$ as*

$$\textsc{Repl}(\mathcal{A}) = \bigcup_{j \geq 1} \left\{ \underbrace{(x, x, \dots, x)}_{j \text{ components}} \mid x \in \mathcal{A} \right\}. \tag{1.7}$$

Proposition 3. *Let $\mathcal{A}$ be a combinatorial class with no element of size* 0 *and let $A(z)$ denote the generating function of $\mathcal{A}$. The generating function of* $\text{Repl}(\mathcal{A})$ *is given by*

$$\sum_{j\geq 1} A(z^j).$$

Finally, we can also note that $\text{MSet}(\mathcal{F} \setminus \mathcal{F}_+) = \mathcal{F}_+ + (\mathcal{F} \setminus \mathcal{F}_+) + \mathcal{E}$ from (1.5). Hence we get the simpler specification of $\mathcal{G}_+$ given by:

$$\mathcal{G}_+ = \text{Repl}(\mathcal{G} \setminus \mathcal{G}_+) \times (\mathcal{F} + \mathcal{E}) \setminus (\mathcal{G} \setminus \mathcal{G}). \tag{1.8}$$

Here again, the specifications (1.6) and (1.8) translate into a system of equations on the generating functions of $\mathcal{G}, \mathcal{G}_+, \mathcal{G}_;, \mathcal{G}_\|$ and $\mathcal{F}$. The analysis of these functions leads to Theorem 3 and is detailed below.

Proof of Theorem 3. In terms of generating functions, the specifications (1.6) and (1.8) of the *annotated* NFJ programs translate into the following system of equations. Note that the equations satisfied by $g_;$ and $g_\|$ are similar to the equations satisfied by $f_;$ and $f_\|$ so we do not repeat the explanations.

$$\begin{aligned}
g(z) &= z + g_;(z) + g_\|(z) + g_+(z) \\
g_;(z) &= \frac{g(z) - g_;(z)}{1 - (g(z) - g_;(z))} = \frac{g(z)^2}{1 + g(z)} \\
g_\|(z) &= \exp\left(\sum_{j\geq 1} \frac{(g - g_\|)(z^j)}{j}\right) - 1 - (g - g_\|)(z) \\
&= g(z) + \ln\frac{1}{1 + g(z)} + \sum_{j\geq 2} \frac{(g - g_\|)(z^j)}{j} \\
g_+(z) &= \left(\sum_{j\geq 1} g(z^j) - g_+(z^j)\right)(1 + f(z)) - (g(z) - g_+(z)).
\end{aligned}$$

Let ρ_g denote the radius of convergence of g. Each program has at least one global choice so there are at least as many elements of size n in $\mathcal{G}$ as in $\mathcal{F}$ and thus $\rho_g \leq \rho$. Using the same argument as above, one can show that $g(z^j)$ is analytic in a disc of radius larger than ρ_g whenever $j \geq 2$. Hence, the rightmost term in the expression of $g_\|$, which we denote by $\zeta_3(z) = \sum_{j\geq 2} \frac{(g-g_\|)(z^j)}{j}$, is analytic in a disc of radius larger than ρ_g. For the same reasons, $\zeta_4(z) = \sum_{j\geq 2} g(z^j) - g_+(z^j)$ is analytic in the same disc and we can write $g_+ = \zeta_4 + \frac{f}{1+f} g$.

Finally, by merging all equations together, we get

$$\begin{aligned}
& g(z) = \zeta_5(z) + (1 + f(z))\psi(g(z)) \\
\text{with} \quad & \psi(z) = \frac{z^2}{1 + z} + z + \ln\frac{1}{1 + z} = \sum_{n\geq 2} \frac{n + 1}{n}(-z)^n \\
\text{and} \quad & \zeta_5(z) = (1 + f(z))(z + \zeta_3(z) + \zeta_4(z)).
\end{aligned}$$

As for f, the key to get the precise behaviour of g near its main singularity, and therefore to get an approximation scheme for ρ_g, is to show that the functional equation $y = y(x,u) = x + u\psi(y)$ has a unique solution, which we are able to describe, and to conclude by unicity that $g(z) = y(\zeta_5(z), 1 + f(z))$. To this end, we use an extension of Theorem 2.21 from [Drm09]. This requires to apply the simple change of variables $\tilde{y} = \frac{y-x}{ux}$ and $\tilde{u} = \frac{u}{1+f(\rho_g)}$ so that the equation fulfils the requirements of the theorem (except for the non-negativity of the coefficients of ψ). The equation then becomes:

$$\tilde{y} = F(x, \tilde{y}, \tilde{u}) = \frac{1}{x}\psi(x(u_1\tilde{u}\tilde{y} + 1)) \quad \text{where } u_1 = 1 + f(\rho_g). \tag{1.9}$$

Remark 2.20 from [Drm09] (which is related to Theorem 2.19 in the book) can actually be adapted to Theorem 2.21. So in order to prove the existence of an analytic continuation of "square-root" type for the unique solution of (1.9), it is enough to show that there exists a pair $(x_1, \tilde{y}_1)$ in the domain of convergence of F such that

$$\begin{aligned} \tilde{y}_1 &= F(x_1, \tilde{y}_1, 1) \\ 1 &= \partial_{\tilde{y}} F(x_1, \tilde{y}_1, 1) \\ 0 &\neq \partial_x F(x_1, \tilde{y}_1, 1) \\ 0 &\neq \partial^2_{\tilde{y}} F(x_1, \tilde{y}_1, 1). \end{aligned}$$

In our case, we have $\partial_{\tilde{y}} F(x, \tilde{y}, \tilde{u}) = u_1\tilde{u}\psi'(x(u_1\tilde{u}\tilde{y} + 1))$ and $\psi'(z) = 2 - \frac{1}{1+z} - \frac{1}{(1+z)^2}$. First we solve $u\psi'(z) = 1$ which yields a unique positive solution $y_1(u) = \frac{2}{\sqrt{1+4(2-u^{-1})}-1} - 1$ which is a decreasing function of u and thus satisfies $y_1(1) > y_1(u) \geq y_1(u_1) \geq y_1(1 + f(\rho))$ for $1 < u \leq u_1$. Note that we have $y_1(1) = \frac{\sqrt{5}-1}{2} \approx 0.618$ and $y_1(1 + f(\rho)) = y_1(\frac{1+\sqrt{3}}{2}) \approx 0.366$.

Thus, a solution $(x_1, \tilde{y}_1)$ of the above system necessarily satisfies $x_1(u_1\tilde{y}_1 + 1) = y_1(u_1)$. Then we solve $\tilde{y}_1 = F(x_1, \tilde{y}_1, \tilde{u})$ by injecting the later equality in the definition of F which yields $x_1(u) = y_1(u) - u\psi(y_1(u))$ in the general case and thus $x_1 = y_1 - u_1\psi(y_1(u_1))$. The two non-nullity conditions are then easily checked since $\partial_x F(x_1, y_1, 1) = (u_1 x_1)^{-1}$ and $\partial^2_y F(x_1, y_1, 1) = x_1 u_1^2 \psi''(y_1(u_1)) > 0$.

We thus obtain that equation (1.9) has a unique solution $\tilde{y}$ which is analytic near $x = 0$ and $u = 1$. Furthermore, this function admits a representation of the following form near $x = x_1$ and $u = 1$ where $\tilde{h}_3$ and $\tilde{h}_4$ are analytic functions:

$$\tilde{y}(x, \tilde{u}) = \tilde{h}_3(x, \tilde{u}) - \tilde{h}_4(x, \tilde{u})\sqrt{1 - \frac{x}{x_1(u_1\tilde{u})}}.$$

From the unicity of the solution, we get that $g(z) = y(\zeta_5(z), 1 + f(z))$ and thus, for some analytic functions $h_3(z)$ and $h_4(z)$, we have:

$$g(z) = h_3(z) - h_4(z)\sqrt{1 - \frac{\zeta_5(z)}{x_1(1 + f(z))}}. \tag{1.10}$$

The singularity ρ_g of g is thus the minimum positive real number in the interval $[0; \rho]$ such that $\zeta_5(\rho_g) = x_1(1 + f(\rho_g))$. Numerically we get that $\rho_g \approx 0.12 < \rho$ so in a neighbourhood of ρ_g we

have

$$g(z) = h_3(z) - h_5(z)\sqrt{1 - \frac{z}{\rho_g}}$$

$$\text{where} \quad h_5(z) = h_4(z)\sqrt{\left(1 - \frac{\zeta_5(z)}{x_1(1+f(z))}\right)\left(1 - \frac{z}{\rho_g}\right)^{-1}} \quad \text{is analytic near } \rho_g.$$

As a consequence, the number g_n of annotated programs is equivalent to $\gamma n^{-\frac{3}{2}}\rho_g^{-n}$ for some constant $\gamma_g > 0$. The numerical evaluation of the constants ρ_g and γ_g is similar to the evaluation of ρ and γ so it is not repeated here. Numerically, we get $\rho_g \approx 0.122854753$. □

This concludes the study of the class NFJ itself and of its number of global choices. The outcome of this sub-section is that we can still obtain an equivalent of the average number of global choices in programs of size n when programs are considered up to commutativity and associativity. Unsurprisingly, this equivalent is of the same form as in the simple case without taking the symmetries into account, though the constants are different. Although it is more satisfactory to take these symmetries into account for quantifying the average number of global choices, this result comes at the expense of a considerably more technical analysis. We now turn to the counting and random sampling of executions until the end of Section 1.2.

1.2.3 The combinatorial toolset part 2: executions as partial increasing labellings

In order to study the set of executions of a program, and in particular in order to count it, we need to give it a combinatorial interpretation too. As mentioned earlier, the idea is to see an execution as a labelling of the actions of the programs. Before formalising this approach, we present a graphical representation of NFJ programs that will help us picture these labellings. This representation yields graphs similar to those of Figure 1.2 on page 13, though with a little more detail so as to remain generic.

Definition 8 (Control graph)**.** *To every NFJ program we associate a control graph with three kinds of nodes: actions* (a)*, fork-join nodes* ($\circ$) *and choices nodes* ($+$)*. The control graph* $G(P)$ *of a program* P *is inductively defined as follows:*

$G(a) = a$

$G(P; Q)$ = $G(P)$ above $G(Q)$, linked by an edge.

$G(P \parallel Q)$ = a $\circ$ node on top with edges to $G(P)$ and $G(Q)$, which are both linked to a $\circ$ node at the bottom.

$G(P + Q)$ = a $+$ node on top with edges to $G(P)$ and $G(Q)$, which are both linked to a $+$ node at the bottom.

As mentioned above, we can see an execution of an NFJ program as a two-step process. First, select a *global choice*, that is select which branch of each choice should be run, and second, label

the actions of this global choice using the integers from $[\![1;n]\!]$ according to the order in which they are fired. The integer n is the number of actions in the global choice here. Figure 1.3 pictures the same example as in Figure 1.2 on page 13 using this representation. In the middle picture, one branch of each + node has been selected and the other has been discarded (coloured in light grey). In the rightmost picture, the remaining actions of the graph have been labelled such that, whenever there is an edge between two actions, the action on the upper end of the edge has a smaller label than the action at the bottom. We call this an increasing labelling of the graph since every path from the top of the graph to the bottom is increasingly labelled.

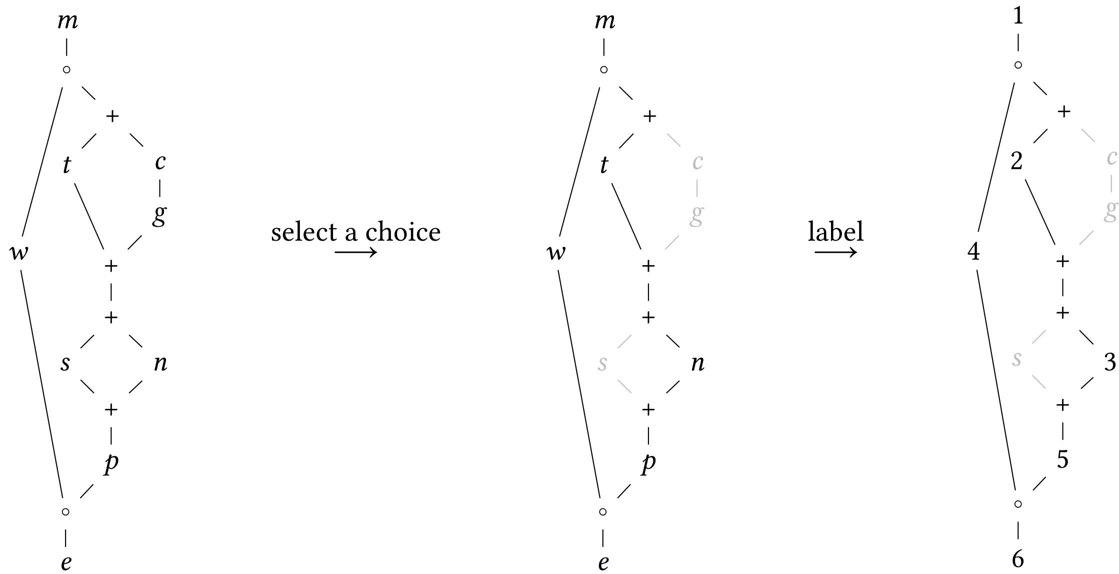


Figure 1.3: Two-step decomposition of an execution of program $P = m; (w \parallel ((t+(c;g)); (s+n); p)); e$

The key idea to study the executions of NFJ programs is to see the set of possible executions of a single program as its own combinatorial class. To this end we will need a *labelled* variant of the formalism presented above. Informally, a labelled combinatorial class is a special case of combinatorial class in which the size n of an object corresponds to a number of "atoms" in the object (typically graph nodes or tree nodes) and where each of these atoms are assigned a unique label from the set $[\![1;n]\!]$. A typical example of labelled class is the set of permutations which can be seen as a linear arrangement of n atoms labelled from 1 to n. In our case, the labelled class of interest is that of the executions of a program, which are seen as a labelled global choice.

More formally, labelled combinatorial classes can be seen as sets of pairs of a regular "unlabelled" object and a permutation representing a labelling.

Definition 9 (Labelled combinatorial class). *Let $\mathfrak{S} = \bigcup_{n\geq 0} \mathfrak{S}_n$ denote the set of all finite permutations. A combinatorial class $\mathcal{C}$ is a* labelled *combinatorial class if it is of the form $\mathcal{C} \subseteq C \times \mathfrak{S}$ for some set C and if for all elements (c, σ) of $\mathcal{C}$ the size $|(c, \sigma)|$ of the element is the unique n such that $\sigma \in \mathfrak{S}_n$.*

As for unlabelled classes, the study of such a class can be made more systematic when one

is able to *specify* it. There exist several operators of the symbolic method which are specific to labelled classes, in addition to those presented in Table 1.2 on page 14. Here we will only need two, the *labelled product* and the *ordered product*. The labelled product (denoted by $\star$) of two objects is defined as the set of all possible interleavings of their respective labellings. Formally, we first defined the labelled product of two objects:

$$(a, \sigma_a) \star (b, \sigma_b) = \{((a, b), \sigma) \mid \sigma \text{ is an interleaving of } \sigma_a \text{ and } \sigma_b\}$$

$$\text{that is} \quad \begin{cases} \sigma \in \mathfrak{S}_{|a|+|b|} \\ \forall i, j \leq |a|, \sigma(i) < \sigma(j) \iff \sigma_a(i) < \sigma_a(j) \\ \forall i, j > |a|, \sigma(i) < \sigma(j) \iff \sigma_b(i) < \sigma_b(j) \end{cases}$$

Said differently, the restriction of σ to $[\![1; |a|]\!]$ is a labelling of a using $|a|$ distinct labels from the interval $[\![1; |a| + |b|]\!]$. Similarly, the restriction of σ to $[\![1 + |a|; |a| + |b|]\!]$ is the labelling of b using the remaining labels. The labelled product of two classes is then defined as a union over all possible pairs:

$$\mathcal{A} \star \mathcal{B} = \bigcup_{a \in \mathcal{A}, b \in \mathcal{B}} a \star b$$

Although the definition is a bit technical, it captures a natural idea in the context of concurrency since it mimics the interleaving semantics of the parallel composition operator. As an example, consider two programs P and Q and two possible executions of these programs e_P and e_Q. Since the rules Lpar and Rpar commute in the semantics given at the beginning of this section, it is easy to see that any interleaving of e_P and e_Q is a valid execution of $P \parallel Q$ and that any execution of $P \parallel Q$ is the interleaving of some executions of P and Q. Hence, if $\mathcal{P}$ and $\mathcal{Q}$ denote the set of all possible executions of P and Q, seen as labelled objects, then $\mathcal{P} \star \mathcal{Q}$ is the set of possible executions of $P \parallel Q$.

The *ordered product* $\mathcal{A} \boxed{\star} \mathcal{B}$ of two labelled classes $\mathcal{A}$ and $\mathcal{B}$ has a simpler definition, although it is less common. Unlike the labelled product, it does not appear in [FS09]. Resources on this operator and its unlabelled counterpart can be found in [BDGP17b]. It is defined by

$$\mathcal{A} \boxed{\star} \mathcal{B} = \left\{ ((a, b), \sigma) \mid (a, \sigma_a) \in \mathcal{A}, (b, \sigma_b) \in \mathcal{B}, \sigma(i) = \begin{cases} \sigma_a(i) & \text{if } i \leq |a| \\ \sigma_b(i - |a|) + |a| & \text{otherwise} \end{cases} \right\}.$$

The ordered product simply shifts the labelling σ_b of its second component b so that it does not overlap with the labelling σ_a of its first component a. More eloquently, in an ordered product, the first component always has the smallest labels and the second component has the largest. In the context of concurrency, this captures the semantics of the sequential composition operator.

In fact, all the constructions of the NFJ language can be mapped to one of the combinatorial constructions we have seen so far. Hence, we can define a *combinatorial specification* of the set of the executions of any program P by induction on the syntax. However, in order to add more information in the specification, which will become useful later for random generation, we introduce a last notion, markers. A marker is a combinatorial class that contains only one object of size 0 and whose purpose is only to distinguish one position in the objects or one subset of the objects (e.g. those that contains the markers by opposition to those that do not). For instance, if

we consider again two NFJ programs P and Q and their respective sets of possible executions $\mathcal{P}$ and $\mathcal{Q}$, the set of the executions of $(P+Q)$ can be specified by $\mathcal{Y}_\ell \star \mathcal{P} + \mathcal{Y}_r \star \mathcal{Q}$ where $\mathcal{Y}_\ell$ (resp. $\mathcal{Y}_r$) is a marker class marking the executions of $(P+Q)$ taking the P branch (resp. the Q branch). In a sense this is a *more precise* specification than $\mathcal{P} + \mathcal{Q}$ since it carries more information.

The function S mapping a program to the specification of its executions is inductively defined in Table 1.3. As a convenience, we assume that all the choices in the program are given a unique identifier i (this is pictured by $+_i$ in all the formulas) so that we can assign them two marker classes $\mathcal{Y}_{i,\ell}$ and $\mathcal{Y}_{i,r}$ marking respectively the executions taking their left branch and their right branch.

Table 1.3: Recursive combinatorial specification of the set of executions of a program and its corresponding generating function

Language construction P	Specification $S(P)$	Generating function $P(z)$
a	$\mathcal{Z}$	z
$P \parallel Q$	$S(P) \star S(Q)$	$P(z) \circledcirc Q(z)$
$P; Q$	$S(P) \boxed{\star} S(Q)$	$P(z)Q(z)$
$P +_i Q$	$\mathcal{Y}_{i,\ell} \times S(P) + \mathcal{Y}_{i,r} \times S(Q)$	$y_{i,\ell}P(z) + y_{i,r}Q(z)$

where $\sum_{n\geq0} a_n z^n \circledcirc \sum_{n\geq0} b_n z^n = \sum_{n\geq0} \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} z^n$.

The generating functions given in the third column of Table 1.3 is the generating function of the executions of the program. It is actually a function of several variables: the main variable z counting the number of atoms in the program and the marker variables $(y_{i,\ell}, y_{i,r}, \ldots)$ marking the different choices. It generalises the generating functions with one variable introduced above so that if $\vec{y}$ is a product of $y_{i,\ell}$ and $y_{i,r}$ variables (where each variable may appear at most once), then the coefficient in front of $z^n\vec{y}$ in the series is the number of execution of size n of the program such that for all i, $y_{i,\ell}$ (resp. $y_{i,r}$) appears in $\vec{y}$ if and only if the left (resp. right) branch of choice number i is taken. In short, the markers encode the local choices while the atomic class $\mathcal{Z}$ encodes the number of actions of the execution.

The operation denoted by $P(z) \circledcirc Q(z)$ is called the coloured product and is introduced in [BDGP17b]. The symbol $\circledcirc$ originally denotes an operation on combinatorial specification and we overload it here to denote an operation on generating functions.

Remark 1. *The reader familiar with combinatorics might find it odd that we use* ordinary *generating functions (OGF) rather than exponential ones (EGF) which are generally more suitable for labelled classes. The reason behind this choice is that we are actually facing operators from both worlds here. The ordered product* $\boxed{\star}$ *expresses in the labelled terms an operation which behaves better in the unlabelled world, and it indeed gives a nice formula in terms of OGF but not in terms of EGF. On the other hand the labelled product* $\star$ *is intrinsically labelled and behaves well only in terms of EGF. Since there is no obvious choice here between the two, we opt for ordinary generating functions for implementation reasons. They require integer arithmetic whereas EGFs would require to deal with rational numbers, which would be less efficient in practice.*

The generating function $P(z)$ of $S(P)$ captures insightful counting information about the program. For instance, the total number of executions of P is obtained by substituting 1 for every variable in $P(z)$. Finer-grained information can also be obtained. For example, given an integer i, the generating function of the subset of the execution of P taking the left branch at choice i is obtained by substituting 1 for $y_{i,\ell}$ and 0 for $y_{i,r}$. The number of such executions can then be obtained by substituting 1 for the remaining variables.

As an example, for the beverage vending machine $P = m; (w \parallel ((t +_1 (c; g)); (s +_2 n); p)); e$, whose control graph is pictured in Figure 1.3 on page 26, we get the specification $S(P) = \mathcal{Z} \boxed{\star} (\mathcal{Z} \star ((\mathcal{Y}_{1,\ell} \star \mathcal{Z} + \mathcal{Y}_{1,r} \star (\mathcal{Z} \boxed{\star} \mathcal{Z})) \boxed{\star} (\mathcal{Y}_{2,\ell} \star \mathcal{Z} + \mathcal{Y}_{2,r} \star \mathcal{Z}) \boxed{\star} \mathcal{Z})) \boxed{\star} \mathcal{Z}$. From this specification we get the following generating function by applying the rules of the symbolic method described in Table 1.3 on the previous page: $P(z) = 4y_{1,\ell}(y_{2,\ell} + y_{2,r})z^6 + 5y_{1,r}(y_{2,\ell} + y_{2,r})z^7$. The number of executions taking the left branch of choice 1, that is choosing tea over coffee, is obtained by substituting 0 for $y_{1,r}$ and 1 for all the remaining variables. This yields 8 whereas the number of executions taking the right branch of choice is 10. This tiny example already shows that sampling executions by choosing one branch of each choice with probability $\frac{1}{2}$ and scheduling the rest of the actions introduces some bias in the generation. While this bias is harmless on such a small example, it can be dramatic in terms of coverage for larger programs as we demonstrate in Section 1.4.

1.2.4 Statistical analysis

We now tackle the problem of exploring the state-space of a given process through random generation. To this end we describe a *uniform* sampler of executions which relies on the counting information contained in the generating function of the program. Our random sampler thus requires the computation of this function as a pre-processing step, which can be done in polynomial time and space. We thus do *not* need to need the explicit, costly construction of the state-space.

Preprocessing: the generating function of executions

As explained in the previous section, the symbolic method gives a systematic way of computing the generating function of the class $S(P)$ of the executions of a program P. However, some care must be taken on the memory representation of this function. Fortunately for us, since the state-space is finite, the generating function of the executions of a program is a polynomial and not an infinite power series, but it has multiple variables encoding the different local choices. We also saw in Theorem 2 that this number of global choices was exponential so fully expanding the generating function of $S(P)$ would yield an exponential number of terms which constrains us to seek a more compact representation.

A more suitable representation is to only expand on the z variable, that is we represent the generating function of $S(P)$ as a dense polynomial in z whose coefficients are arithmetic expressions stored as trees sharing some common sub-structures. More precisely, an arithmetic expression is a binary tree whose internal nodes store a flag indicating whether the node corresponds to an addition or a multiplication, and whose leaves are either a pair of the form (i, s) indicating a $y_{i,s}$ variable or an integer. Moreover, the implementation of these coefficients must use hash-consing (see [Got74]). That is to say that when an expression (resulting from previous

computations) is used several times, it should not by copied but referenced by a pointer. Note that this is different from optimal compaction where common sub-terms are systematically compacted. Finally, the generating function of $S(P)$ is stored as an array of such coefficients such that the coefficient at position i is the coefficient of degree i in z. An example of such a polynomial is pictured in Figure 1.4.

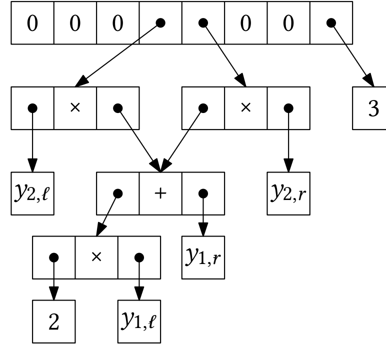


Figure 1.4: The compact tree-based memory representation of $y_{2,\ell}(2y_{1,\ell} + y_{1,r})z^3 + y_{2,r}(2y_{1,\ell} + y_{1,r})z^4 + 3z^7$. Note that the common sub-term $(2y_{1,\ell} + y_{1,r})$ is shared by two expressions.

A straightforward application of the symbolic method leads to Algorithm 2 for computing the generating function of the executions of a program. In the algorithm, the multiplications and additions of two coefficients are implemented as the allocation of a new tree node whose two children are the two operands.

Algorithm 2 Computation of the generating function of the executions of an NFJ program

Input: An NFJ program P
Output: The generating function of $S(P)$
  function GFUN(P)
    if $P = a$ **then return** z
    else if $P = (Q \parallel R)$ **then return** GFUN(Q) $\circledcirc$ GFUN(R)
    else if $P = (Q; R)$ **then return** GFUN(Q) $\cdot$ GFUN(R)
    else if $P = (Q +_i R)$ **then return** $y_{i,\ell}$GFUN(Q) + $y_{i,r}$GFUN(R)

where $\sum_{n\geq 0} a_n z^n \circledcirc \sum_{n\geq 0} b_n z^n = \sum_{n\geq 0} \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} z^n$.

The coloured product $\circledcirc$ used in the parallel composition case can be implemented similarly to the "text-book" polynomial multiplication using the formula given in the algorithm. Note however that computing each binomial coefficient individually incurs a non-negligible cost since they require big-integer arithmetic and cannot be obtained in constant time. This cost can be reduced significantly by re-using the value of the last-computed binomial coefficient at each iteration step using the formula $\binom{n}{k+1} = \binom{n}{k}\frac{n-k}{k+1}$. The whole procedure is presented in Algorithm 3 and the trick used to compute the binomial coefficient faster is at line 7. This technique allows to lower the cost of computing one binomial coefficient to only one multiplication and one division of a big integer by a small integer fitting in a machine word.

Algorithm 3 Computation of the coloured product of two polynomials

Input: Two polynomials P and Q of respective degrees d_P and d_Q
Output: The coloured product of P and Q

```
1: function ⊚(P, Q)
2:     R ⟵ array of length d_P + d_Q + 1
3:     for n from 0 to d_P + d_Q + 1 do
4:         b ⟵ 1
5:         c ⟵ P[0] · Q[n]
6:         for k from 1 to n do                ▷ Invariant: upon entering the loop b = (n choose k−1)
7:             b ⟵ b · (n − k + 1)/k
8:             c ⟵ c + P[k] · Q[n − k] · b
9:         R[n] ⟵ c
10:    return R
```

Theorem 5 (Complexity of Algorithm 2)**.** *Algorithm 2 can be implemented in complexity* $O(|P|^2)$ *in terms of memory allocations and arithmetic operations on big integers, where* $|P|$ *denotes the number of atomic actions in* P*.*

Proof. For all programs P, let $C(P)$ denote the number of arithmetic operations on *coefficients* performed by GFUN(P). All the binary operators of NFJ have a cost that is at most of the order of one polynomial multiplication. Moreover, note that the degree of the generating function of P (which, recall, is a polynomial) is at most $|P|$. Hence, there exists a constant $\alpha \geq 1$ such that for all P, Q we have $C(P; Q), C(P + Q), C(P \parallel Q) \leq 2\alpha|P||Q| + C(P) + C(Q)$.

We prove by induction on the syntax of programs that $C(P) \leq \alpha|P|^2$. This is trivially true when $P = a$ and if $\bullet$ denotes any of the three other operators, we have $C(P \bullet Q) \leq 2\alpha|P||Q| + \alpha(|P|^2 + |Q|^2) \leq \alpha(|P| + |Q|)^2 = \alpha|P \bullet Q|^2$.

Note that it is crucial to use hash-consing for this results to hold. Said differently, one arithmetic operation on expressions must only consist in the allocation of a new tree node and must *not* perform any deep copy. □

1.2.5 Random sampling of executions

Our random sampling algorithm builds on ideas from the so-called *recursive method*, which is due to Nijenhuis and Wilf [NW78]. We use a two-step process to generate uniform executions. First, we select one of the global choices of the program with probabilities depending on the number of labellings of each global choice. Second, once a global choice has been selected, we draw one of the possible labellings of this choice uniformly at random. In other words, at the first step we *bias* our generation of a global choice so that the overall process remains uniform in terms of executions.

In order to sample a global choice, we first choose the *size* of the choice to be sampled. To this end, recall that the coefficient of degree k of the generating function encodes all executions of length k and that they correspond to global choices of size k. Besides, the coefficient

of degree k of the generating function can be seen as the generating function of the size-k executions of the program. Thus, substituting 1 for every variable occurring in it yields the total number of size-k executions. The size of the global choice to be sampled must therefore be chosen with a probability proportional to the full evaluation of its corresponding coefficient. This is implemented in Algorithm 4. For example, for the beverage vending machine, we have $P(z) = 4y_{1,\ell}(y_{2,\ell} + y_{2,r})z^6 + 5y_{1,r}(y_{2,\ell} + y_{2,r})z^7$ which has two coefficients. Substituting 1 for all y variables yields $P(z) = 8z^6 + 10z^7$, which tells us that the global choice to be sampled must have size 6 with probability $\frac{8}{8+10} = \frac{4}{9}$ and size 7 with probability $\frac{5}{9}$.

Algorithm 4 Random sampling of a coefficient of the generating function of P

Input: The generating function $P(z)$ of the executions of a program P
  function SAMPLE_SIZE($P(z)$)
    for $c \cdot z^d \in P(z)$ **do**
      $W_d \longleftarrow$ EVAL_COEFF(c)
    $m \longleftarrow$ RAND_UNIF($[1; \sum W_d]$)
    $d \longleftarrow$ the minimum index d s.t. $m \leq \sum_{d' \leq d} W_{d'}$
    return d

Now, recall that a coefficient is an arithmetic expression encoded as a tree whose internal nodes are sums (+) or products (×). Once a coefficient has been selected, we traverse this coefficient recursively, from top to bottom and, select only one child of each sum node we encounter, and collect y variables along the way. The idea is to construct a global choice from these y variables since each one of them encodes a local choice. More precisely, at each sum node $e_1 + e_2$ in the traversal, we compute the total number of executions, $e_1(1)$ and $e_2(1)$, encoded by both terms and we choose the term i (for $i \in \{1, 2\}$) with probability $\frac{e_i(1)}{e_1(1)+e_2(1)}$. Conversely, at each product node we traverse recursively both children since they contribute to two “parts” of the same executions whereas the children of a sum node contribute to two disjoint sets of executions. In the end, the set of y variables that we have seen in the process are interpreted as follows: $y_{i,\ell}$ corresponds to choosing the left branch of choice i and $y_{i,r}$ corresponds to choosing the right branch. This is described in more detail in Algorithm 5 which returns a list of y variables.

The function EVAL_COEFF used in both Algorithms 4 and 5 fully evaluates an expression by substituting 1 for all its variables. It is given in Algorithm 6 for the sake of completeness.

Finally, once the local choices returned by Algorithm 5 are applied to the program, that is to say that all the unused branches are removed, there remains to sample a uniform execution of the remaining choice-free program. This has been covered in [BDGP17a] which proposes, in particular, an algorithm that is optimal in terms of random bits. We do not recall this algorithm here. The complete procedure for sampling a uniform execution in P is given in Algorithm 7. Naturally, the pre-processing step at line 2 must only be done once if one wishes to sample several executions for the same program.

If we set aside the complexity of the pre-processing step, which has already been covered in Theorem 5, most of the remaining cost of the generation is hidden in the SAMPLE_SIZE and SAMPLE_CHOICE functions. First, these functions need to evaluate the coefficients of the generating functions. Since these evaluations, as well as the evaluations of some of their sub-terms, are to

Algorithm 5 Random sampling of a global choice of a given size

Input: An arithmetic expression e
Output: A list of local choices

```
function SAMPLE_CHOICE(e)
    if e = e1 + e2 then
        p1 ⟵ EVAL_COEFF(e1)
        p2 ⟵ EVAL_COEFF(e2)
        if BERNOULLI(p1/(p1+p2)) then return SAMPLE_CHOICE(e1)
        else return SAMPLE_CHOICE(e2)
    else if e = e1 × e2 then return CONCAT(SAMPLE_CHOICE(e1), SAMPLE_CHOICE(e2))
    else if e = y_{i,s} then return [y_{i,s}]
    else if e = e′/n then return SAMPLE_CHOICE(e′)
    else if e = n then return []                                  ▷ empty list
```

Algorithm 6 Full evaluation of an expression

Input: An arithmetic expression e encoded as a tree
Output: An integer

```
function EVAL_COEFF(e)
    if e = e1 + e2 then return EVAL_COEFF(e1) + EVAL_COEFF(e2)
    else if e = e1 × e2 then return EVAL_COEFF(e1) · EVAL_COEFF(e2)
    else if e = y_{i,s} then return 1
    else if e = n then return n
```

Algorithm 7 Full procedure for the uniform sampling of executions in NFJ

Input: an NFJ program P
Output: a uniform execution of P

```
1: function SAMPLE_EXEC(P)
2:     P(z) ⟵ GFUN(P)
3:     k ⟵ SAMPLE_SIZE(P(z))
4:     e ⟵ the coefficient of degree k of P(z)
5:     y⃗ ⟵ SAMPLE_CHOICE(e)
6:     P′ ⟵ apply the global choice y⃗ to P
7:     return SAMPLE_CHOICEFREE(P′)          ▷ See [BDGP17a] for SAMPLE_CHOICEFREE
```

be re-used later in the generation process, they must be cached using a dynamic programming approach. This implies that the memory layout presented in Figure 1.4 on page 30 should be adapted to reserve some space for one big-integer in each tree node. Second, these functions need to traverse a whole coefficient and to draw Bernoulli random variables. Since the evaluation and caching part must only be done once too, it can be performed during the pre-processing step. Moreover, each node of each coefficient incurs one arithmetic operation on big integers, so the complexity of this part of the algorithm in terms of arithmetic operations is of the same order as the space complexity of Algorithm 2. In addition, the traversal of a coefficient requires

a similar number of memory accesses and a linear number of calls to the generator of Bernoulli variables. Theorem 6 summarises these remarks and is the main result of this section.

Theorem 6 (complexity of the random sampling algorithm)**.** *Sampling uniform random executions of a program P requires:*

- *a pre-processing step of complexity $O(|P|^2)$ in terms of memory allocations and arithmetic operations on big integers;*
- *the generation of a linear number of Bernoulli variables and $O(|P|^2)$ memory accesses.*

Moreover, all the big integers at play here are bounded by $|P|!$ so their bit-size is bounded by $|P|\log_2|P|$.

Proof. The complexity of the pre-processing and of the random generation have already been discussed and only the bit-size of the integers remains to prove. All the integers we manipulate in the algorithms are bounded by the maximum possible number of executions of a program. A straightforward induction shows that for any programs P and Q, the total number of executions of $(P;Q)$ and of $(P+Q)$ is upper-bounded by the total number of executions of $(P \parallel Q)$. Hence, the maximum possible number of executions of a program of size n is obtained when the program is made only of atomic actions and parallel compositions, which corresponds to $n!$. □

1.2.6 Experimental study

In order to assess experimentally the efficiency of our method, we put into use the algorithms presented here and demonstrate that they can handle systems with a significantly large state space. We generated a few NFJ programs at random using a Boltzmann random generator. All the polynomial operations and coefficients were implemented in OCaml.

Note that we did not optimize our code for efficiency nor ran extensive benchmarking, hence the numbers we give should be taken as a rough estimate of the performance of our algorithms. For the sake of reproducibility, the source code of our experiments is available on a Gitlab repository[2]at `https://gitlab.com/ParComb/libnfj`.

Table 1.4 on the following page reports the runtime of the preprocessing phase (Algorithm 2), the runtime of the random sampler (Algorithm 7) and the number of executions of various programs. For the runtime of the counting algorithm, every measurement was performed 7 times and we reported the median of these 7 values. For the random sampler, every measure was performed 101 times and for each one we report the median of these values as well as the interquartile range (IQR)[3], which gives an idea of the dispersion of the measures. We use these metrics rather than the mean and the variance to reduce the importance of extremal values and give a precise idea of what runtime the user should expect when running our sampler. The time reported is the CPU time. The state-space column indicates the total number of executions. Finally, the mem. size column reports the amount of memory occupied by the generating function of executions computed by GFUN.

---

[2]All the benchmarks were run on a standard laptop with an Intel Core i7-8665U and 32G of RAM running Ubuntu 20.10 with kernel version 5.8.0-48-generic. We used OCaml version 4.08.1 and GMP version 6.2.0.

[3]The interquartile range of a set of measures is the difference between the third and the first quartiles. Compared with the value of the median, it gives a rough estimate of the dispersion of the measures.

Table 1.4: Quick benchmark of the counting and random sampling functions of executions for loop-free programs

size	# executions	mem. size	GFUN	UNIFEXEC	IQR
100	$1.168 \cdot 2^{108}$	65.30K	0.000091s	0.010ms	0.001ms
200	$1.956 \cdot 2^{199}$	235.12K	0.000245s	0.022ms	0.001ms
500	$1.249 \cdot 2^{645}$	2.21M	0.004563s	0.091ms	0.007ms
1000	$1.012 \cdot 2^{903}$	5.92M	0.011524s	0.135ms	0.008ms
2000	$1.354 \cdot 2^{2381}$	50.40M	0.076030s	0.429ms	0.093ms
3000	$1.682 \cdot 2^{6331}$	591.75M	0.987996s	1.562ms	0.309ms
5000	$1.464 \cdot 2^{10085}$	1.92G	2.959532s	3.239ms	0.413ms

1.3 Extending the model with loops

This section is devoted to extending our model with loops. This is a significant improvement in terms of expressiveness, but has major implications on the state-space of programs.

First, programs may now have an infinite number of executions, and as a consequence, there is no *uniform* distribution over their executions any more. To circumvent this issue, we turn to the uniform generation of executions of a given length n where n is given as an input of the problem. Such a sampler can also be used, in conjunction with a particular procedure to select a length at random according to some particular distribution, for instance to sample uniform executions of length at most n.

A second, more significant, consequence of adding loops is that it interacts with the non-deterministic choices, as a choice may occur inside a loop and thus may be duplicated multiple times as we unroll the loop. Thus, the notion of global choice we have defined in the previous section, allowing us to decide of *all* the choices at once and then executing the rest of the program, does not extend well in the presence of loops. In a way, by introducing loops, we trade the clean separation we had between the non-determinism and the interleaving semantics of NFJ for more expressiveness.

As a consequence, we must take another approach to random sampling in this section. We will use the structure of the program to guide the generation rather than the structure contained in a multi-variate generating function as before.

1.3.1 Non-deterministic fork-join processes with loops

We start by extending the model from the previous section with a loop construction expressing that a program may be executed any number of times. Also, the empty program 0 used in the semantics of Section 1.2 as a convenience, is now usable in the syntax. The complete updated grammar of NFJ terms is given below. Note that only the two last lines are new.

$$
\begin{array}{llll}
P, Q & ::= & P \parallel Q & \text{parallel composition} \\
& | & P; Q & \text{sequential composition} \\
& | & P + Q & \text{non-deterministic choice} \\
& | & a \in \mathcal{A} & \text{atomic action} \\
& | & P^\star & \text{loop} \\
& | & 0 & \text{empty program}
\end{array}
$$

As in the previous section, programs are considered up to alpha-equivalence and atomic actions are assumed to occur only once within a term. Again, since we only model the control-graphs of concurrent processes, the loop construction expresses that the body of the loop may be executed any number of times but does not state under which condition we exit the loop. Informally, the loop $P^\star$ may have either no iteration, in which case it behaves as 0, or at least one, in which case it behaves as $(P; P^\star)$. We introduce the empty program in the grammar here, not only as a convenience, but also because it provides a slight gain of expressiveness, as it allows to write program such as $(0 + P)$ which express optional computations.

The semantics of programs must be updated to express the behaviour of loops. We first define a nullable predicate which indicates whether a program may terminate without firing any action. We can start the next iteration of a loop only if the current iteration is nullable.

$$
\begin{array}{ll}
\text{nullable}(P \parallel Q) = \text{nullable}(P) \wedge \text{nullable}(Q) & \text{nullable}(0) = \text{true} \\
\text{nullable}(P; Q) = \text{nullable}(P) \wedge \text{nullable}(Q) & \text{nullable}(a) = \text{false} \\
\text{nullable}(P + Q) = \text{nullable}(P) \vee \text{nullable}(Q) & \text{nullable}(P^\star) = \text{true}
\end{array}
$$

The reduction relation $P \xrightarrow{a} P'$ introduced in the first section is then extended to loops. Note that we also need to modify the reduction rule for the sequential composition. Since we now have non-empty programs which may terminate without firing any action (the nullable programs), we now want to allow the right-hand-side of a sequence to start its execution whenever the left-hand-side is nullable. The full new list of reduction rules is given below in Figure 1.5.

$$
\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}\ (\text{Lpar}) \qquad
\frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}\ (\text{Rpar}) \qquad
\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q}\ (\text{Lseq})
$$

$$
\frac{\text{nullable}(P) \quad Q \xrightarrow{a} Q'}{P; Q \xrightarrow{a} Q'}\ (\text{Rseq}) \qquad
\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\ (\text{Lchoice}) \qquad
\frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\ (\text{Rchoice})
$$

$$
\frac{}{a \xrightarrow{a} 0}\ (\text{act}) \qquad
\frac{P \xrightarrow{a} P'}{P^\star \xrightarrow{a} P'; P^\star}\ (\text{loop})
$$

Figure 1.5: Semantic of NFJ with loops

We call “execution step” a *proof-tree* built from the above rules and we define an execution as a sequence of such steps leading to a nullable term.

Definition 10 (Execution). *An execution of an NFJ program P_0 is a sequence of steps of the form $P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \dots \xrightarrow{a_n} P_n$, such that* nullable($P_n$) *holds, and where for all i, $P_{i-1} \xrightarrow{a_i} P_i$ is a* proof-tree*, that is it contains all the applied rules and not simply its conclusion.*

We refer to the set of all possible executions of a program as its state-space.

As an example the program $a^{\star\star}$ has two executions of length 2, both firing a twice. One corresponds to the case where the outer loop is only unrolled once (i.e. the (*loop*) rule is only applied once) but the inner loop twice. The other corresponds to the case where the outer loop is unrolled twice and the two occurrences of the inner loop once. The first step of both executions is the same and is depicted below:

$$\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} 0}\,(\text{act})}{a^\star \xrightarrow{a} 0; a^\star}\,(\text{loop})}{a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{loop})$$

Then the second step of the two executions are the following. On the left, the inner loop fires a second a while, on right, the first iteration of the inner loop terminates (we apply the (Rseq) rule at the top level) and a second iteration of the outer loop starts.

$$\dfrac{\dfrac{\text{nullable}(0) \qquad \dfrac{\dfrac{}{a \xrightarrow{a} 0}\,(\text{act})}{a^\star \xrightarrow{a} 0; a^\star}\,(\text{loop})}{0; a^\star \xrightarrow{a} 0; a^\star}\,(\text{Rseq})}{(0; a^\star); a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{Lseq}) \qquad \dfrac{\text{nullable}(0; a^\star) \qquad \dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} 0}\,(\text{act})}{a^\star \xrightarrow{a} 0; a^\star}\,(\text{loop})}{a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{loop})}{(0; a^\star); a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{Rseq})$$

We will take the following program as a running example for the rest of the section: $P_0 = ((a + (b \parallel c))^\star \parallel (d + 0))^\star; (e + (f \parallel g))$. This program has one length-1 execution firing only e and four length-2 executions respectively firing fg, gf, ae and de.

1.3.2 Combinatorial interpretation

From a combinatorial point of view, the introduction of loops in the language has two levels of implication.

Syntax At the syntactic level first, it is still possible to interpret the set of NFJ programs as a combinatorial class but this requires to choose a more adequate notion of size. Recall that a combinatorial class is given by a set of objects and a size function such that there is a finite number of objects of size n for all integers n. The number of actions contained in a program is not eligible as a size function since there is an infinite number of (syntactic) programs with one

action, for instance $a, a^\star, a^{\star\star}$, etc. A more adequate notion of size is the number of constructors used to build a program, that is

$$\begin{aligned} &|P;Q|_c = |P+Q|_c = |P \parallel Q|_c = 1 + |P|_c + |Q|_c \\ &|P^\star|_c = 1 + |P|_c \\ &|a|_c = |0|_c = 1. \end{aligned} \tag{1.11}$$

We use the notation $|P|_c$ to denote this new size function in order to avoid confusion with the number of atoms of a program $|P|$. Using, this size function, the class $\mathcal{F}$ of NFJ programs can be specified by

$$\begin{aligned} \mathcal{F} = \ &\mathcal{Z} \times \mathcal{F} \times \mathcal{F} \\ &+ \mathcal{Z} \times \mathcal{F} \times \mathcal{F} \\ &+ \mathcal{Z} \times \mathcal{F} \times \mathcal{F} \\ &+ \mathcal{Z} \\ &+ \mathcal{Z} \times \mathcal{F} \\ &+ \mathcal{Z}. \end{aligned} \tag{1.12}$$

Two differences must be noted compared to Section 1.2. First, the size function is different so that both the atomic action and the empty program have size 1 (hence the two $\mathcal{Z}$), and each constructor "costs" one $\mathcal{Z}$ since they are counted in the size. The second difference is that we have two more terms, one for the empty program $\mathcal{Z}$, and one for loop terms $\mathcal{Z} \times \mathcal{F}$. From this specification, we get that the generating function f of NFJ programs satisfies $f(z) = z(2 + f(z) + 3f(z)^2)$. This equation can be solved explicitly and we get

$$f(z) = \frac{1 - z - \sqrt{(1-z)^2 - 24z^2}}{6z}. \tag{1.13}$$

By studying the behaviour of this function near its main singularity $\rho = (1 + 2\sqrt{6})^{-1}$, we can obtain the asymptotic number of program of size n using the transfer theorem from [FS09]. See the proof of Theorem 1 in the previous section for a similar but more detailed proof. This leads to the following result.

Theorem 7. *The number of NFJ programs (with loops) of size n is equivalent to $Cn^{-\frac{3}{2}}\rho^{-n}$ when n tends to the infinity, where $\rho^{-1} = 1 + 2\sqrt{6} \approx 5.898979$ and $C = \frac{1}{18\sqrt{\pi}}\sqrt{4 + \sqrt{2/3}} \approx 0.068789$.*

This can be compared with Theorem 1 using the property that a binary tree with n leaves has $n - 1$ internal nodes. This implies that if a program is built only using the constructors from the previous section and has n atomic actions, then it is made of $2n-1$ constructors. Thus, when n denotes the number of *constructors*, the exponential factor in Theorem 1 becomes $\sqrt{12}^n$ (for odd values of n) and $\sqrt{12} \approx 3.464 < 5.899$.

Executions As in the previous section, we define a specification $S(P)$ of the class of the executions of a program P. Some differences must be noted with the previous section. First, we need a new operator, the *ordered set* operator, modelling a sequence of increasingly labelled objects,

which expresses the semantics of the loop. Like the *ordered product* $\boxed{\star}$, this is an uncommon operator, which has been studied in [BDGP17b]. Another difference is that the presence of the loop and the empty program makes possible for non-trivial programs to have the empty execution as a valid execution. Although this seems innocuous, it forces us to handle carefully some special cases in our specification to avoid counting the same execution twice. Finally, the generating function of the executions of $S(P)$ which we refer to as the generating function of the executions of P is not a polynomial any more, but is an infinite (but convergent) formal power series. The recursive definition of $S(P)$ and its generating function are given in Table 1.5. A detailed explanation of the different constructions is given below.

Table 1.5: Simplified recursive combinatorial specification of the set of executions of a program and its corresponding generating function

Language construction P	Specification $S(P)$	Generating function $P(z)$
0	$\mathcal{E}$	1
a	$\mathcal{Z}$	z
$P \parallel Q$	$S(P) \star S(Q)$	$P(z) \odot Q(z)$
$P; Q$	$S(P) \boxed{\star} S(Q)$	$P(z)Q(z)$
$P + Q$ when nullable$(P) \wedge$ nullable(Q)	$S(P) + (S(Q) \setminus \mathcal{E})$	$P(z) + Q(z) - 1$
$P + Q$ otherwise	$S(P) + S(Q)$	$P(z) + Q(z)$
$P^\star$ when nullable(P)	$\textsc{OSet}(S(P) \setminus \mathcal{E})$	$(1 - (P(z) - 1))^{-1}$
$P^\star$ otherwise	$\textsc{OSet}(S(P))$	$(1 - P(z))^{-1}$

where $\sum_{n\geq 0} a_n z^n \odot \sum_{n\geq 0} b_n z^n = \sum_{n\geq 0} \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} z^n$.

The empty program 0 and the atomic action a have only one execution, of length 0 and 1 respectively. This is modelled combinatorially by the neutral class $\mathcal{E}$ — the class containing only one element of size 0 — and the atom class $\mathcal{Z}$ — the class with only one element of size 1.

As before, the executions of $P \parallel Q$ are made of any interleaving of one execution of P and one execution of Q. For instance if $P = a + (b; c)$ and $Q = d^\star$, then P admits for instance an execution firing b and then c (denoted by bc for short) and Q admits an execution firing two ds (denoted by dd for short). Then all the 6 possible interleavings of these executions are executions of $P \parallel Q$: $bcdd$, $bdcd$, $bddc$, $dbcd$, $dbdc$ and $ddbc$ (again, we only denote the executions by their firing sequences for conciseness). Although the interpretation of these interleavings using increasing labellings is less obvious than is the previous section, the notion at play here is still well captured by the labelled product of combinatorics, denoted by $\star$.

The executions of $P; Q$ are given by an execution of P followed by an execution of Q. So for instance, using the same example programs P and Q as above, $bcdd$ is an execution of $(P; Q)$ but not $dbcd$. So they can be seen as a pair of an execution of P and an execution of Q, which is still expressed using the ordered product $\boxed{\star}$.

The set of executions of $P + Q$ is the union of the executions of P and Q. Moreover this union is "almost" disjoint in the sense that the only execution that these programs may have in common is the empty execution, hence the two cases in the definition. Combinatorially, the fact

that nullable(P) holds corresponds to the fact that the class of its executions contains one object of size 0, the empty execution. It is in fact important that we can express this in terms of *disjoint* unions because they fit in the framework of analytic combinatorics whereas arbitrary unions are more difficult to handle[4].

Finally, the executions of $P^\star$ are sequences of executions of P or, equivalently, sequences of *non-empty* executions of P. This second formulation leads to a non-ambiguous specification as the unique class $\mathcal{P}'$ satisfying $\mathcal{P}' = \mathcal{E} + \mathcal{P}_+ \boxed{\star}\, \mathcal{P}'$, where $\mathcal{P}_+$ denotes the non-empty executions of P. This implicitly defined class $\mathcal{P}'$ is denoted $\text{OS}\textsc{et}(\mathcal{P}_+)$ and is called the *sequence* of $\mathcal{P}_+$. Once again we must distinguish whether nullable(P) holds or not in the definition of $\mathcal{P}_+$ to avoid ambiguities and thus double-counting.

The S function described above maps each program to a combinatorial specification of its executions. As an example, for our example program P_0 introduced above, we have $S(P_0) = \text{OS}\textsc{et}(\text{OS}\textsc{et}(\mathcal{Z} + (\mathcal{Z} \star \mathcal{Z})) \star (\mathcal{Z} + \mathcal{E}) \setminus \mathcal{E}) \boxed{\star} (\mathcal{Z} + (\mathcal{Z} \star \mathcal{Z}))$. The generating function of the executions of a program, i.e. of the class $S(P)$, constitutes a condensed summary of the counting information of its state space. Our uniform random sampler of executions for NFJ programs with loops will use the generating function of each sub-term of P to generate an execution of P.

Before diving into the description of our random sampler, we want to give another example of application of analytic combinatorics, by showing how a few manipulations on polynomials can lead to interesting algorithmic applications and precise quantitative results. We already showed in Theorems 1 and 2 how to get the asymptotic number of programs and the average number of global choices of loop-free programs using this kind of techniques. Here, we study the generating function $P_0(z)$ of the example program P_0 given above, which we recall here for convenience: $P_0 = [(a + (b \parallel c))^\star \parallel (d + 0)]^\star; [e + (f \parallel g)]$. Let $P_0(z) = \sum_{n\geq 0} p_n z^n$ denote the expansion in power series of the generating function of $S(P_0)$ and recall that its n-th coefficient p_n is the number of executions of P_0 of length n. By applying the rules from Table 1.5 on the previous page we obtain that:

$$\begin{aligned} P_0(z) &= \left[(1 - z - 2z^2)^{-1} \circledcirc (z+1)\right]^{-1} \cdot [z + 2z^2] \\ &= \frac{(2z+1)(2z-1)^2(z+1)^2 z}{1 - 4z - 4z^2 + 6z^3 + 8z^4}. \end{aligned}$$

The second line of the above formula is obtained by applying the calculus rule[5] $z \circledcirc A(z) = z\frac{d(zA(z))}{dz}$. From this formula we derive two applications. First, from the denominator of this rational expression we deduce that for all $n > 6$ we have $p_n - 4p_{n-1} - 4p_{n-2} + 6p_{n-3} + 8p_{n-4} = 0$. The obtained recurrence formula can be used to compute the number of executions of length n of P_0 in linear time. On the analytic side, $P_0(z)$ being a rational function, we can do a *partial fraction decomposition* to obtain $P_0(z)$ as a sum of four terms of the form $C_i(1 - z\rho_i^{-1})$ (plus a polynomial). Each of these terms expands as $\sum_{n\geq 0} C_i \rho_i^{-n} z^n$, hence the number of executions of P_0 of length n satisfies $p_n = C \cdot \rho^{-n} \cdot (1 + o(1))$ for some constants C and ρ and with an exponentially

[4] Grammar descriptions involving non-disjoint unions are referred to as “ambiguous” and lack most of the benefits, if not all, of the symbolic method, essentially because some objects may be counted multiple times when applying the method.

[5] This is the only “non-standard” computation rule we use in this example. All the rest is usual polynomial manipulations. Some more rules for computing $A(z) \circledcirc B(z)$ will be given in Section 1.4.2.

small error term hidden in the $o(1)$. In this case we have $\rho \approx 0.221987$, $C \approx 0.146871$ and the error term is of the order of 0.327950^n. Table 1.6 compares the values of p_n — computed using the aforementioned linear algorithm — and of the proposed approximation for a few values of n. One can see that already for small values of n, the relative error of this approximation is rather low.

Table 1.6: Value of p_n, of its approximation $C \cdot \rho^{-n}$ and of the relative error $|p_n - C \cdot \rho^{-n}|/p_n$ for small values of n

n	6	7	8	9	10	…	30
p_n	1226	5528	24904	112196	505424	…	5985551205783341568
approx	1227	5529	24907	112199	505429	…	5985551205783353055
rel. err.	8.16e-4	1.81e-4	1.2e-4	2.67e-5	9.89e-6	…	1.92e-15

1.3.3 Statistical analysis algorithms

We now study the problem of exploring the state-space of a given program through random generation. In the presence of loops, programs can have an infinite number of executions and there is thus no uniform distribution over executions. However, programs still have a finite number of executions of a given length, so we propose a uniform sampler of executions of fixed length. This sampler can be used to target executions of specific length n or, in conjunction with a procedure to sample a size, to sample a uniform execution of length bounded by n for instance. In Section 1.4, we will see an alternative approach for exploring the state-space, based on the generation of execution *prefixes*.

Preprocessing: the generating function of executions

As explained above, the symbolic method gives a systematic way of computing the generating function of the class of the executions of a program P from its specification $S(P)$. The computation rules are given in Table 1.5 on page 39. Since we are in presence of infinite spate-spaces, these generating functions are not polynomials any more and become (convergent) formal power series. For the algorithms to remain practical, we only compute the $(n + 1)$ first terms of these series, hence allowing us to sample uniform executions of length k for all $k \leq n$. Algorithm 8 implements the computation of the n first terms of the generating functions of *all* sub-terms of a program. The resulting (partial) generating functions must be stored in the tree representation of the program.

The coloured product $\circledcirc$ used in the parallel composition case can be implemented using the "naive" algorithm as in the previous section. There is a more efficient approach here though, because we are using integer coefficients rather than expressions. The idea is to use the combinatorial Laplace and Borel transforms to express this operation as a regular product. This is achieved by the formula $A \circledcirc B = \mathcal{L}(\mathcal{B}(A) \cdot \mathcal{B}(B))$, where the Borel transform $\mathcal{B}$ is defined by $\mathcal{B}(\sum_{n\geq 0} a_n z^n) = \sum_{n\geq 0} \frac{a_n}{n!} z^n$ and the Laplace transform $\mathcal{L}$ is defined by $\mathcal{L}(\sum_{n\geq 0} a_n z^n) = \sum_{n\geq 0} n! a_n z^n$. More details

Algorithm 8 Computation of the generating function of the executions of an NFJ program, and all its sub-terms, up to degree n

Input: An NFJ program P and a positive integer n.
Output: The first $n+1$ terms of the generating function of P

```
function GFUN(P, n)
    if P = 0 then return 1
    else if P = a then return z
    else if P = Q ∥ R then return GFUN(Q, n) ⊚ GFUN(R, n) mod z^{n+1}
    else if P = Q; R then return GFUN(Q, n) · GFUN(R, n) mod z^{n+1}
    else if P = Q + R then
        q(z) ⟵ GFUN(Q, n),     r(z) ⟵ GFUN(R, n)
        return q(z) + r(z) − q(0)r(0)
    else if P = Q^⋆ then
        q(z) ⟵ GFUN(Q, n)
        return (1 − (q(z) − q(0)))^{−1} mod z^{n+1}
```

on the coloured product, the Borel and Laplace transform and their applications can be found in [BDGP17b].

To be implemented efficiently using only integer rather than rational arithmetic, the coefficients of the result of the Borel transform should share $n!$ as a common denominator where n is the degree of the polynomial and the division by $n!$ should be postponed to the last moment. So if $A(z) = \sum_{k=0}^{n} a_k z^k$ and $B(z) = \sum_{k=0}^{m} b_k z^k$, then

$$\mathcal{B}(A) = \frac{1}{n!}\sum_{k=0}^{n} \frac{n!}{k!} a_k z^k = \frac{1}{n!}\tilde{A}(z) \qquad \mathcal{B}(B) = \frac{1}{m!}\sum_{k=0}^{m} \frac{m!}{k!} b_k z^k = \frac{1}{m!}\tilde{B}(z)$$

and

$$A(z) \circledcirc B(z) = \frac{1}{n!m!}\mathcal{L}(\tilde{A}(z) \cdot \tilde{B}(z)).$$

Thus, the coloured product can be implemented as in Algorithm 9 where the polynomial multiplication at line 13 is where most of the computational cost lies. The advantage of this approach is that it leaves the choice of the polynomial multiplication algorithm open and we can thus benefit from existing fine-tuned implementations of the algorithms from the literatures. The FLINT library [9] for instance provides such algorithms.

The computation of $(1 - (q(z) - q(0)))^{-1}$ at the last line of Algorithm 8 can be carried out efficiently using the so-called Newton method (see [VG13, p. 259] and [PSS12] for instance). The algorithm consists in computing the sequence $S_{i+1}(z) \longleftarrow S_i(z) + S_i(z) \cdot ((q(z) - q(0)) \cdot S_i(z) - (S_i(z) - 1))$, starting fvom $S_0(z) = 1$, and until the $(n+1)$ first coefficients of $S_i(z)$ are the same as those of $(1 - (q(z) - q(0)))^{-1}$. The intuition behind this formula comes from the field of numerical analysis where the Newton method is used to get fast-converging approximations of real constants.

The key feature of this approximation scheme is that the error term, that is the difference between $S_i(z)$ and its limit $(1 - (q(z) - q(0)))^{-1}$, is squared at each iteration. As a consequence, the number of correct terms in $S_i(z)$ doubles at each iteration and only a logarithmic number of iterations of the formula is necessary to compute the $(n+1)$ first terms of the solution. To make

Algorithm 9 Fast implementation of the coloured product for integer polynomials

Input: Two integer polynomials $A(z)$ and $B(z)$ of respective degrees n and m and stored as arrays of integers
Output: The coloured product $A(z) \circledcirc B(z)$ of A and B

```
function COLPROD(A, B)
    Ã, B̃ ⟵ copies of A and B
    f ⟵ 1
    for k from n down to 1 do
        Ã[k] ⟵ Ã[k] · f
        f ⟵ f · k
    Ã[0] ⟵ Ã[0] · f
    g ⟵ 1
    for k from m down to 1 do
        B̃[k] ⟵ B̃[k] · g
        g ⟵ g · k
    B̃[0] ⟵ B̃[0] · g
    R ⟵ Ã · B̃
    for k from 0 to m + n do R[k] ⟵ R[k]/(f · g)
    return R
```

this argument more formal, let $\tilde{q}(z) = q(z) - q(0)$, let $S(z) = (1 - \tilde{q}(z))^{-1}$ and let $E_i(z)$ denote the error term of $S_i(z)$, that is $E_i(z) = S(z) - S_i(z)$. By observing that $S(z) = 1 + \tilde{q}(z)S(z)$, we have that

$$\begin{aligned} S_{i+1}(z) &= S_i(z) + S_i(z)(S_i(z)\tilde{q}(z) - (S_i(z) - 1)) \\ &= S_i(z) + S_i(z)(S(z) - 1 - E_i(z)\tilde{q}(z) - S(z) + 1 + E_i(z)) \\ &= S(z) - E_i(z) + (S(z) - E_i(z))E_i(z)(1 - \tilde{q}(z)) \\ &= S(z) - (1 - \tilde{q}(z))E_i(z)^2 \\ \text{hence } E_{i+1} &= (1 - \tilde{q}(z))E_i(z)^2. \end{aligned}$$

Since $E_0(0) = 0$, we have that the first term of the expansion of $E_0(z)$ is zero and by induction the 2^i first terms of $E_i(z)$ are zero. As a consequence, the 2^i first terms of $S_i(z)$ are the same as the 2^i first terms of $S(z)$. Algorithm 10 implements this scheme. Note that in the formula at line 5, it is only necessary to compute the two products up to degree $2i$.

Assume we have a so-called multiplication function $\mathcal{M} : \mathbb{N} \longrightarrow \mathbb{N}$ as defined in the "generalities" Chapter on page 6. The following lemma expresses the complexity of Algorithm 10 as a function of $\mathcal{M}(n)$.

Lemma 2. *Algorithm 10 can be implemented in $O(\mathcal{M}(n))$ arithmetic operations on integers.*

Proof. At each iteration of the loop, two multiplication of polynomials of degree $2i$ are performed (the terms of higher degrees can be safely ignored). Thus, the total cost of the multiplications is $2\mathcal{M}(n)+2\mathcal{M}(n/2)+2\mathcal{M}(n/4)+2\mathcal{M}(n/8)\cdots \leq 2\mathcal{M}(2n) = O(\mathcal{M}(n))$. The additions only contribute to a lower order term. □

Algorithm 10 Computing the $n+1$ first terms of the quasi inverse of a polynomial using the Newton method

Input: A polynomial $q(z)$
Output: The $n+1$ first terms of $(1-q(z)+q(0))^{-1}$ as a polynomial of degree n

```
1: function QINV(q)
2:     S(z) ⟵ 1
3:     i ⟵ 1
4:     while i < n + 1 do
5:         S(z) ⟵ S(z) + S(z)(S(z)(q(z) − q(0)) − (S(z) − 1))
6:         i ⟵ 2i
7:     return S(z)
```

Theorem 8. *Let P be an NFJ program and let $|P|_c$ denote its syntactic size as defined in (1.11). Algorithm 8 can be implemented to compute the first n coefficients of the generating function of the executions of P in $O(|P|_c\mathcal{M}(n))$ operations on big integers, where $\mathcal{M}(n)$ is the complexity of the multiplication of two polynomials of degrees at most n.*

Proof. The proof of Theorem 8 follows from the above discussion: each constructor incurs one polynomial operation among addition, multiplication, coloured product and quasi-inverse and all of them can be carried out in $O(\mathcal{M}(n))$. □

An important question which complements Theorem 8 is that of the cost of one arithmetic operation. Since this cost is a function of the bit-size of the integers, this can be reformulated into: what is the size of the integers at play? Theorem 9 gives an upper bound on these coefficients. This upper bound is expressed using the *height* of a program, that is its maximum number of nested operators, which is recursively defined by

$$\begin{aligned} h(a) &= h(0) = 0 \\ h(P \parallel Q) &= h(P+Q) = h(P;Q) = 1 + \max(h(P), h(Q)) \\ h(P^\star) &= 1 + h(P). \end{aligned}$$

Theorem 9. *Let P be an NFJ program and let $n \geq 0$. The number p_n of length-n executions of P is at most $2^{h(P)n}$ and its bit-size $\lceil \log_2(p_n) \rceil$ is thus bounded by $h(P)n$.*

Proof. This upper bound can be proven by induction on P.

- It is trivially true for the base cases $P = 0$ and $P = a$.
- The number of length-n executions of $(P;Q)$ is upper-bounded by the number of length-n executions of $(P \parallel Q)$, which is itself bounded, by induction hypothesis, by

$$\sum_{k=0}^{n} \binom{n}{k} 2^{h(P)k + h(Q)(n-k)} = (n+1)2^{\max(h(P),h(Q))n} \leq 2^{(1+\max(h(P),h(Q)))n}.$$

- For $n \geq 1$, the number of length-n executions of $(P+Q)$ is bounded by induction by $2^{h(P)n} + 2^{h(Q)n} \leq 2^{h(P+Q)n}$.

- Finally, if p_i denotes the number of executions of length i of P, then the number of executions of $P^\star$ is given by

$$\begin{aligned}\sum_{k=1}^{n}\sum_{\substack{i_1,i_2,\dots,i_k>0\\ i_1+i_2+\cdots+i_k=n}} p_{i_1}p_{i_2}\cdots p_{i_k} &\leq \sum_{k=1}^{n}\sum_{\substack{i_1,i_2,\dots,i_k>0\\ i_1+i_2+\cdots+i_k=n}} 2^{h(P)(i_1+i_2+\cdots+i_k)}\\ &= \sum_{k=1}^{n}\sum_{\substack{i_1,i_2,\dots,i_k>0\\ i_1+i_2+\cdots+i_k=n}} 2^{h(P)n}\\ &= 2^{h(P)n}\cdot 2^{n-1} \leq 2^{h(P^\star)n}\end{aligned}$$

□

To give a rough idea of the performance that can be achieved by Algorithm 8, we computed the generating function of P_0 up to degree $n = 10000$ — and thus its number of executions of length k for all $k \leq 10000$ — in less that 4s on a standard PC. A more detailed benchmark of Algorithm 8 is given in Section 1.3.4.

Random sampling of executions

In order to sample a uniform execution directly from the syntax of the program, we use the so-called “recursive method”, as introduced in [NW78] and integrated into the analytic combinatorics framework in [FZV94]. It operates in a similar fashion to the symbolic method, that is by induction on the specification, by combining the random samplers of the sub-structures with simple rules depending on the grammar construction. For the sake of clarity we represent executions as sequences of atomic actions in the presentation of the algorithm. This encoding does not contain all the information that defines an execution, typically it does not reflect in which iteration of a loop an atomic action is fired for instance. However it makes the presentation clearer and the algorithm can be easily adapted to a more faithful encoding. Our uniform random sampler of executions is described in Algorithm 11 and the detailed explanations about the different constructions are given below.

Choice The simplest rule of the recursive method is that of the disjoint union used at line 4 of Algorithm 11. If q_n and r_n denote the number of length-n executions of Q and R, then a uniform random length-n execution of $P = Q + R$ is a uniform length-n execution of Q with probability $q_n/(q_n + r_n)$ and a uniform length-n execution of R otherwise. One way to draw the Bernoulli variable is to draw a uniform random big integer x in $[\![0; q_n + r_n[\![$ and to return `true` if and only if $x < q_n$. As an example, consider the programs $Q = (a + (b \parallel c))$ and $R = d^\star$. We count that Q has two executions of length two: bc and cb and R has only one: dd. Hence, to sample a length-2 execution in $(Q + R)$, one must perform a recursive call on Q with probability 2/3 and on R with probability 1/3.

Parallel composition The other rules build on top of the disjoint union case. For instance, the set of length-n executions of $P = Q \parallel R$ can be seen as $\mathcal{Q}_0 \star \mathcal{R}_n + \mathcal{Q}_1 \star \mathcal{R}_{n-1} + \cdots + \mathcal{Q}_n \star \mathcal{R}_0$ where $\mathcal{Q}_k$ (resp. $\mathcal{R}_k$) denotes the set of length-k executions of Q (resp. R). By generalising the previous rule to disjoint unions of $(n + 1)$ terms, and using the fact that the number of elements of $\mathcal{Q}_k \star$

Algorithm 11 Uniform random sampler of executions of given length

Input: A program P and an integer n such that P has length n executions.
Output: A list of atomic actions representing an execution

```
1:  function UnifExec(P, n)
2:      if n = 0 then return the empty execution
3:      else if P = a then return a
4:      else if P = Q + R then
5:          if Bernoulli(q_n/(q_n+r_n)) then return UnifExec(Q, n)
6:          else return UnifExec(R, n)
7:      else if P = Q ∥ R then
8:          draw k ∈ ⟦0; n⟧ with probability (n choose k) q_k r_{n−k} / p_n
9:          return shuffle(UnifExec(Q, k), UnifExec(R, n − k))
10:     else if P = Q; R then
11:         draw k ∈ ⟦0; n⟧ with probability q_k r_{n−k} / p_n
12:         return concat(UnifExec(Q, k), UnifExec(R, n − k))
13:     else if P = Q* then
14:         draw k ∈ ⟦1; n⟧ with probability q_k p_{n−k} / p_n
15:         return concat(UnifExec(Q, k), UnifExec(P, n − k))
```

The lower case letters p_n, q_k, r_{n-k} etc. indicate the number of executions of length n, k, $n-k$ of programs P, Q and R.

$\mathcal{R}_{n-k}$ is $q_k r_{n-k}\binom{n}{k}$, one can select in which one of these terms to sample by drawing a random variable which is k with probability $q_k r_{n-k}\binom{n}{k}/p_n$. Then it remains to sample a uniform element of $\mathcal{Q}_k$, a uniform element of $\mathcal{R}_{n-k}$ and a uniform shuffling of their labellings among the $\binom{n}{k}$ possibilities. This is described at line 7 of Algorithm 11. We do not detail the implementation of the shuffling function here, an optimal algorithm in terms of random bits consumption, can be found in [BDGP17a]. As an example, consider the same programs as above: $Q = (a + (b \parallel c))$ and $R = d^\star$. The number of length-3 executions of $(Q \parallel R)$ is $1 \cdot 1 \cdot \binom{3}{1} + 2 \cdot 1 \cdot \binom{3}{1} = 9$ using the decomposition $Q_1 \star R_2 + Q_2 \star R_1$. Say $k = 1$ is selected (with probability 1/3), then the recursive calls to $(Q, 1)$ and $(R, 2)$ necessarily return a and dd and the SHUFFLE procedure must choose a shuffling uniformly between add, dad and dda.

Sequential composition The case of the sequential composition is similar (see line 10 of Algorithm 11). We use the same kind of decomposition, using the Cartesian product $\times$ in place of the labelled product $\star$. This has the consequence of removing the binomial coefficient in the formula for the generation of the k random variable. Once k is selected, we generate an execution of Q_k, an execution of R_{n-k} and we concatenate the two.

Loop Finally, the case of the loop is a slight adaptation of the case of the sequential composition using the fact that the executions of $Q^\star$ are the executions of $(0 + Q; Q^\star)$. However, care must be taken to avoid issues related to double-counting. More specifically, when sampling an execution of $(Q; Q^\star)$ we must not choose an execution of length 0 for the left-hand-side Q. This is related

to the same reason we had to specify the executions of $Q^\star$ as all the sequences of *non-empty* executions of Q. This is presented at line 13 of Algorithm 11, note that $k > 0$. As an example, for sampling a length-3 execution in $(a + (b; c))^\star$, one may select $k = 1$ with probability 2/3, which yields abc or aaa depending on the recursive call to $(Q^\star, 2)$ or $k = 2$, with probability 1/3, which yields bca.

Generation of random variables We did not give details on how to generate the random variable k for the parallel, sequential and loop case. In has been shown in [FZV94; Mol05; MM04] that good performance can be achieved by using the so-called boustrophedonic order. For instance, in the case of the sequential composition $P = (Q; R)$, the idea is to generate a random integer x in the interval $[\![0; p_n[\![$ and to find the minimum number ℓ such that the sum of ℓ terms $q_0 r_n + q_n r_0 + q_1 r_{n-1} + q_{n-1} r_1 + q_2 r_{n-2} + \cdots$ (taken in this particular order) is greater than x. Then k is such that the last term of this sum is $q_k r_{n-k}$. The key idea of this algorithm is that the worst case of this algorithm corresponds to choosing k close to $n/2$ which yields a divide and conquer scheme.

Theorem 10. *Using the boustrophedonic order, the complexity of the random generation of an execution of length n in P in terms of arithmetic operations on big integers is* $O(n \cdot \min(\ln(n), h(P)))$ *where* $h(P)$ *refers to the* height *of* P*.*

Contrary to the classical context of random generation that we have in analytic combinatorics (like in [FZV94; MM04; Mol05]), the grammar enumerating the executions to be sampled is not a constant but rather a parameter of the problem. Hence its size cannot be considered constant and the complexity analysis needs to be carefully crafted to take this variable into account.

Proof. The $O(n \ln(n))$ bound follows from Theorem 11 of [FZV94]. We obtain the other bound by refining the result of Theorem 12 from [Mol05].

The combinatorial classes we are considering are built from the $\star, \times, +$ and $\textsc{Seq}(\cdot)$ operators without recursion, they hence fall under the scope of *iterative classes* for which Molinero proved a linear complexity in n. However the proof given in [Mol05] does not give an explicit bound for the multiplicative constants, which actually depend on the size of the grammar and which we cannot consider constant in our context. Let $C(P, n)$ denote the cost of $\textsc{UnifExec}(P, n)$ in terms of arithmetic operations on big integers. We show that $C(P, n) \leq \alpha n h(P)$ by induction for some constant α to be specified later.

- The base cases have a constant cost.
- The case of the choice only incurs a constant number c of arithmetic operations in addition to the cost of the recursive calls. Hence $C(Q+R, n)$ is bounded by $c+\alpha \max(C(Q, n), C(R, n))$. By induction, this quantity is then bounded by $c + \alpha n \max(h(Q), h(R)) = c + \alpha n(h(Q + R) - 1)$ and thus, if $\alpha \geq c$, then $C(Q + R, n) \leq \alpha n h(Q + R)$.
- The parallel composition case incurs a number of arithmetic operations of the form $c' \cdot \min(k, n - k)$ where k is the random variable generated using the boustrophedonic order technique. Hence $C(Q \parallel R, n)$ is bounded by $c' \min(k, n - k) + C(Q, k) + C(R, n - k)$ and by induction by $c' \min(k, n - k) + \alpha k h(Q) + \alpha(n - k)h(R) \leq \alpha n h(Q \parallel R) + c' \min(k, n - k) - \alpha n$. The last term on the right is bounded by 0 if $\alpha \geq c'$.

- Sequential composition is treated using the same argument as for parallel composition.
- Finally, the loop must be handled by reasoning "globally" on the total number of unrollings. Say the loop $Q^\star$ is unrolled r times. Then its cost is bounded by $c' \sum_{i=1}^{r} \min(k_i, k_{i+1} + \cdots + k_r) + \sum_{i=1}^{r+1} C(Q, k_i)$. The first sum is bounded by $c'n$ and the second is bounded by induction by $\sum_{i=1}^{r+1} \alpha k_i h(Q)$ which simplified to $\alpha n h(Q)$. Hence, reusing the bound $\alpha \geq c'$ and the fact that $h(Q^\star) = 1 + h(Q)$, we get $C(Q^\star, n) \leq \alpha n h(Q^\star)$ which terminates the proof. □

1.3.4 Experimental study

In order to assess experimentally the efficiency of our method, we put into use the algorithms presented here and demonstrate that they can handle systems with a significantly large state space. We generated a few NFJ programs at random using a Boltzmann random generator. In its basic form, the Boltzmann sampler would generate a high number of loops and a large number of sub-terms of the form $P + 0$ in the programs which we believe is not realistic so we tuned it using [BBD18] so that the number of both types of nodes represent only 10% of the size of the program in expectation. We rely on the FLINT library (Fast Library for number theory [9]) to carry all the computations on polynomials except for the coloured product and the quasi-inversion using Newton iteration which we implemented ourselves. The former was not provided natively by the library and the latter was feasible using FLINT's primitives but slow compared to the Newton method.

Note that besides the choice of the algorithms, we did not optimize our code for efficiency nor ran extensive benchmarking, hence the numbers we give should be taken as a rough estimate of the performance of our algorithms. For the sake of reproducibility, the source code of our experiments is available on a Gitlab repository[6]at `https://gitlab.com/ParComb/libnfj`.

Table 1.7 on the following page reports the runtime of the preprocessing phase (Algorithm 8), the runtime of the random sampler (Algorithm 11) and the number of executions of length n of various programs of various values of n. For the runtime of the counting algorithm, every measurement was performed 7 times and we reported the median of these 7 values. For the random sampler, every measure was performed 101 times and for each one we report the median of these values as well as the interquartile range (IQR)[7], which gives an idea of the dispersion of the measures. We use these metrics rather than the mean and the variance to reduce the importance of extremal values and give a precise idea of what runtime the user should expect when running our sampler. The time reported is the CPU time as measured by C's `clock` function. The state-space column indicates the number of executions of length n. Finally, the mem. size column reports the amount of memory occupied by the generating functions of executions computed by GFUN.

1.4 Execution prefixes generation

In this section, we describe a complementary tool to explore the state-space of a program: a uniform random sampler of execution *prefixes*. Execution prefixes offer a more fine-grained tool

[6] All the benchmarks were run on a standard laptop with an Intel Core i7-8665U and 32G of RAM running Ubuntu 20.10 with kernel version 5.8.0-48-generic. We used FLINT version 2.6.3-2 and GMP version 6.2.0

[7] The interquartile range of a set of measures is the difference between the third and the first quartiles. Compared with the value of the median, it gives a rough estimate of the dispersion of the measures.

Table 1.7: Quick benchmark of the counting and random sampling functions of executions

size	len	# executions	mem. size	GFUN	UNIFEXEC	IQR
100	500	$1.370 \cdot 2^{1119}$	898.98K	0.015s	0.241ms	0.020ms
100	1000	$1.690 \cdot 2^{2234}$	3.32M	0.053s	0.521ms	0.059ms
500	500	$1.071 \cdot 2^{1589}$	2.84M	0.087s	0.359ms	0.069ms
500	1000	$1.093 \cdot 2^{3102}$	10.38M	0.538s	1.094ms	0.133ms
1000	500	$1.096 \cdot 2^{2374}$	8.75M	0.289s	0.496ms	0.068ms
1000	1000	$1.579 \cdot 2^{4756}$	33.19M	1.645s	1.842ms	0.177ms
2000	500	$1.336 \cdot 2^{2273}$	10.42M	0.355s	0.914ms	0.293ms
2000	1000	$1.551 \cdot 2^{4624}$	39.20M	1.890s	1.119ms	0.128ms

to explore the state-spaces as a sampler of prefixes can be combined with other tools or with custom heuristics to bias the random generation toward regions of interest of the state space. We see this algorithm as a building block to construct exploration strategies and the fact that it is *uniform* over prefixes of a given length implies that is still gives *control* over the distribution of the sampled values.

Note that sampling a uniform prefix of a given length is different from sampling a uniform execution of larger length and truncating it. For instance, the program $P = a^\star + (b + c)^\star$ has three execution prefixes of length 1, namely a, b and c but the probability that an execution of length n has a as a prefix is only $1/(1 + 2^n)$. The prefix a will thus statistically never appear among executions of large length. This trivial example illustrates that in order to achieve a good coverage on the set of prefixes of a given length of a program, a dedicated sampler is necessary. At the end of this section we will compare our sampler experimentally to another classical random sampling technique, called isotropic sampling, and used for instance in [GS05].

We start by defining formally the notion of prefix. Then we apply the methodology that we have developed in the previous sections to tackle the problem of the uniform generation of prefixes: specify the objects to be sampled, count them and use the counting information to sample. We also show a quantitative result on the number of prefixes of a program and its relation to the number of full executions.

Definition 11 (execution prefixes)**.** *An execution prefix of an NFJ program P is any, possibly empty, sequence of executions steps starting from P of the form $P \overset{a_1}{\Rightarrow} P_1 \overset{a_2}{\Rightarrow} P_2 \cdots \overset{a_n}{\Rightarrow} P_n$. Note that P_n is not necessarily* nullable *here.*

1.4.1 Specification of the prefixes

As for the uniform random generation of execution in the previous sections, we start by re-formulating the problem into combinatorial terms. Just like we have described a combinatorial specification $S(P)$ of the class of the executions of a program P, we describe here how to compute a combinatorial specification $S_p(P)$ of the class of the execution prefixes of P. Interestingly, this specification can be seen as the specification of the executions of a new program $\text{pref}(P)$ whose *full executions* are in bijections with the *execution prefixes* of P. More eloquently, for each pro-

gram P, we can define a new program $\text{pref}(P)$ such that $S_p(P) = S(\text{pref}(P))$. The recursive rules used to compute $\text{pref}(P)$ as well as $S_p(P)$ and its generating function are given in Table 1.8. The combinatorial interpretation of each rule is detailed below.

Table 1.8: Recursive rules for the computation of (1) the program $\text{pref}(P)$ whose executions are in bijection with the execution prefixes of P, (2) the specification $S_p(P)$ of the execution prefixes of P and (3) its generating function $\bar{P}(z)$

Prog. P	Prefix program $\text{pref}(P)$	Specification $S_p(P)$	Generating function $\bar{P}(z)$
0	0	$\mathcal{E}$	1
a	$0 + a$	$\mathcal{E} + \mathcal{Z}$	$1 + z$
$P \parallel Q$	$\text{pref}(P) \parallel \text{pref}(Q)$	$S_p(P) \star S_p(Q)$	$\bar{P}(z) \circledcirc \bar{Q}(z)$
$P + Q$	$\text{pref}(P) + \text{pref}(Q)$	$S_p(P) + (S_p(Q) \setminus \mathcal{E})$	$\bar{P}(z) + \bar{Q}(z) - 1$
$P; Q$	$\text{pref}(P) + (P; \text{pref}(Q))$	$S_p(P) + S(P) \boxed{\star} \left(S_p(Q) \setminus \mathcal{E}\right)$	$\bar{P}(z) + P(z)(\bar{Q}(z) - 1)$
$P^\star$	$P^\star; \text{pref}(P)$	$\mathcal{E} + S(P^\star) \boxed{\star} (S_p(P) \setminus \mathcal{E})$	$1 + \frac{\bar{P}(z)-1}{1-(P(z)-P(0))}$

First of all, remark that our definition of execution prefixes includes the empty prefix, having zero execution steps, as well as all the full executions of the program. So for all P, we must have $\mathcal{E} \subset S_p(P)$ and $S(P) \subset S_p(P)$.

For instance, the program a consisting of only one atomic action has two execution prefixes, the empty prefix and the prefix firing a. Hence, its set of prefixes is modelled by $\mathcal{E} + \mathcal{Z}$ which also corresponds to the executions of the program $0 + a$. The empty program only has the empty prefix.

Another simple case is that of the parallel composition. The execution prefixes of $P \parallel Q$ are exactly all the possible interleavings of a prefix of P and a prefix of Q. Hence, its set of prefixes is $S_p(P) \star S_p(Q)$ which corresponds to the executions of $\text{pref}(P) \parallel \text{pref}(Q)$. Similarly, the prefixes of $P + Q$ are simply the union of the respective prefixes of P and Q. Moreover, the intersection of these two sets always contains exactly one element, the empty prefix. Hence the prefixes of $P + Q$ are unambiguously specified by $S_p(P) + (S_p(Q) \setminus \mathcal{E})$, which corresponds to the executions of $\text{pref}(P) + \text{pref}(Q)$. Recall that combinatorial specifications need to be unambiguous for the symbolic method, that is the set of rules dictating how to compute the generating function of $S_p(P)$, to apply.

The case of the sequential composition is more interesting. We distinguish between the prefixes of executions of $P; Q$ which only fire actions from P, and those which have fired at least one action from Q. Said differently, the former correspond to the prefixes which always use the (Lseq) rule of the semantics to $P; Q$ and the latter correspond to those which use the (Rseq) rule at some point. The prefixes firing only actions from P are specified by $S_p(P)$ and the ones firing at least one element from Q are made of a full execution of P followed by a non-empty prefix from Q, that is $S(P) \boxed{\star} (S_p(Q) \setminus \mathcal{E})$. Note that it is necessary to only consider non-empty prefixes of Q in this second case, so as to ensure that the two specifications do not overlap. The program $\text{pref}(P) + (P; \text{pref}(Q))$ has the same executions.

Finally, the case of the loop is a generalisation of the above reasoning. All the prefixes of $P^\star$,

at the exception of the empty prefix, are made of any sequence of *full* non-empty executions of P, corresponding to full iterations of the loop, followed by a non-empty prefix of P, corresponding to the last, possibly partial, iteration. Note that a sequence of non-empty executions of P is actually an execution of $P^\star$.

With this specification at hand, we can derive two kinds of results on the set of execution prefixes of a program. First, we show that the number of execution prefixes of length n of a program is of the same order as its number of executions of length n (provided it has at least one such execution). This means that prefixes are, in average, shared by many executions. Second, on the algorithmic side, we describe a uniform random sampler of execution prefixes in the same fashion as the random sampler of execution of Section 1.3.

1.4.2 Quantitative analysis

The number of execution prefixes of length n of a program is trivially lower-bounded by its number of executions of length n. A natural question to ask is how many more prefixes than executions a program has. In this sub-section we quantify the number of prefixes of programs precisely and we prove that, in most cases, the number of prefixes of length n of a program is asymptotically of the same order as its number of executions of length n. We describe the different possible configurations.

The results of this section build on the following technical result, which states that the generating functions of the executions and of the prefixes have the same asymptotic behaviour.

Theorem 11. *Let P be an NFJ program containing at least one loop. Let $P(z)$ denote its generating function of executions and let $\bar{P}(z)$ denote its generating function of execution prefixes. We have that $P(z)$ and $\bar{P}(z)$ have the same radius of convergence $0 < \rho \leq 1$. Furthermore, there exist three constants $\bar{C} \geq C > 0$ and $\alpha \in \mathbb{N}^*$ such that near ρ we have $P(z) \sim \frac{C}{(\rho-z)^\alpha}$ and $\bar{P}(z) \sim \frac{\bar{C}}{(\rho-z)^\alpha}$.*

Note that a program has an infinite number of executions if and only if it contains at least one loop (the pattern $0^\star$ being forbidden). We thus only reason on programs having an infinite state-space here. Since the generating function $P(z)$ (resp. $\bar{P}(z)$) is rational, its coefficients can be written as a linear combination of geometric terms of the form r^n, with polynomial coefficients, where r is a pole of $P(z)$ (resp. $\bar{P}(z)$).

$$[z^n]P(z) = \sum_{i=1}^{k} p_i(n) r_i^n \qquad \deg(p_i) = \text{degree of the pole } r_i \text{ in } P(z)$$

$$[z^n]\bar{P}(z) = \sum_{i=1}^{\ell} \bar{p}_i(n) \bar{r}_i^n \qquad \deg(\bar{p}_i) = \text{degree of the pole } \bar{r}_i \text{ in } \bar{P}(z)$$

From Theorem 11, we also know that the dominant terms in both sequences are of the same order $n^{\alpha-1}\rho^{-n}$. So the behaviour of the number of executions *and* and the number of prefixes of length n should be of the order of $n^{\alpha-1}\rho^{-n}$. But, because these functions might have several poles of modulus ρ, there might be periodic compensations in these sequences.

For instance, the program $P = (a \parallel b)^\star$ only has executions of even length so that its number of executions is $p_n = \mathbb{1}_{\{2|n\}} \sqrt{2}^n$. However, its number of prefixes is $\bar{p}_n = \mathbb{1}_{\{2|n\}} \sqrt{2}^n + 2\mathbb{1}_{\{2|n-1\}} \sqrt{2}^{n-1}$,

which is thus of the order of p_n only for even value of n. A precise description of the possible behaviours of such sequences is given in [FS09, Theorem V.3] but we only focus on a corollary here, which is more insightful in our context.

Corollary 1. *Let P be an NFJ program with an infinite state-space (thus with at least one loop), let P_n denote its number of executions of length* at most *n and let $\bar{P}_n$ denote its number of executions prefixes of length* at most *n. There exists a constant $\lambda > 0$ such that we have $P_n \leq \bar{P}_n \leq \lambda P_n$. Moreover, if α and ρ denote the constants from Theorem 11, we have that $P_n = \Theta(n^{\alpha-1}\rho^{-n})$ if $\rho < 1$ and $P_n = \Theta(n^{\alpha})$ otherwise.*

Considering the number of executions of length at most n has advantage of mitigating the periodic effects described above and thus describes more faithfully the growth rate of the state-space. This corollary establishes that, in the sense given above, a program has roughly the same number of execution prefixes as it has executions.

The rest of the sub-section is dedicated to the proof of Theorem 11. The proof of Theorem 11 is done by induction on the syntax of the program and is actually straightforward, except for the case of the coloured product, for which we must establish some analytical properties. Lemma 3 gives a calculus formula for the coloured product of a polynomial with any function and Lemmas 5 and 4 gives formulas for computing the coloured product of any two rational functions. We only characterise the properties of the coloured product over rational functions here since the generating function of the executions and the prefixes of an NFJ program are necessarily rational.

Lemma 3. *Let $A(z) = \sum_{n\geq 0} a_n z^n$ be a formal power series and let k be a non-negative integer. We have the identity $z^k \circledcirc A(z) = \frac{z^k}{k!}\frac{d^k}{dz^k}(z^k A(z))$, which holds both formally and analytically in the domain of convergence of A.*

Proof. By definition we have that $z^k \circledcirc A(z) = \sum_{n\geq 0} a_n \binom{n+k}{k} z^{n+k}$. Moreover, the k-th derivative $\frac{\mathrm{d}^k}{\mathrm{d}z^k}(z^{n+k})$ of z^{n+k} is given by $(n+k)(n+k-1)\cdots(n+1)z^n = \binom{n+k}{k}k!z^n$, hence the result of the lemma. □

Using Lemma 3, we obtain that if A is a rational function, then $z^k \circledcirc A(z)$ has the same poles as A and these poles have the same degree as in A, incremented by k. In Lemmas 5 and 4 below we give two formulas allowing to compute the product of any two rational functions.

Lemma 4. *Let a be a complex number, we have that $(1+az)^k \circledcirc \dfrac{1}{1-az} = \dfrac{1}{(1-az)^{k+1}}$.*

Before going over the proof of this lemma, observe that this formula has a combinatorial interpretation, in terms of rational languages, when a is an integer. Let Σ be an alphabet with a letters and let $\Sigma_1, \Sigma_2, \dots, \Sigma_k$ be k distinct copies of Σ. The left-hand-side of the equality given in the lemma is the generating function of the rational language L_1 obtained as the shuffle of $(\epsilon + \Sigma_1).(\epsilon+\Sigma_2)\dots(\epsilon+\Sigma_k)$ with $\Sigma^\star$. The right-hand-side of this equality is the generating function of the language $L_2 = \Sigma_1^\star.\Sigma_2^\star \dots \Sigma_k^\star.\Sigma^\star$. The equality of the generating functions is explained by the following bijection between the two languages. A word w in L_1 can be uniquely decomposed

as $w = w_{i_1} u_{i_1} w_{i_2} u_{i_2} \cdots w_{i_\ell} u_{i_\ell} w'$ where $0 \le \ell \le k$, $w_{i_1}, \dots, w_{i_\ell}, w' \in \Sigma^\star$ and $u_{i_j} \in \Sigma_{i_j}$ for all j. Since Σ_{i_j} is a copy of Σ, w_{i_j} can be mapped to a unique word w'_{i_j} in Σ_{i_j} and thus w can be mapped to a unique word $w'_{i_1} u_{i_1} w'_{i_2} u_{i_2} \cdots w'_{i_\ell} u_{i_\ell} w'$ in L_2. Furthermore, all the words of L_2 can be obtained in such a way. The general case is given by a computational proof.

Proof of Lemma 4. By definition we have

$$(1 + az)^k \circledcirc \frac{1}{1 - az} = \sum_{n \ge 0} \sum_{j=0}^{n} \binom{n}{j} \binom{k}{j} a^j a^{n-j} z^n = \sum_{n \ge 0} \left(\sum_{j=0}^{n} \binom{n}{j} \binom{k}{k-j} \right) (az)^n.$$

And by Vandermonde's identity $\sum_{j=0}^{n} \binom{n}{j}\binom{k}{k-j} = \binom{n+k}{k}$, we have

$$\sum_{n \ge 0} \left(\sum_{j=0}^{n} \binom{n}{j} \binom{k}{k-j} \right) (az)^n = \sum_{n \ge 0} \binom{n+k}{k} (az)^n = \left(\frac{1}{1 - az} \right)^{k+1}. \qquad \square$$

Lemma 4 allows to decompose a pole of any degree as a polynomial and a simple pole. We need one last identity allowing to compute the coloured product of two simple poles.

Lemma 5. *Let a and b be two complex numbers, we have* $\frac{1}{1 - az} \circledcirc \frac{1}{1 - bz} = \frac{1}{1 - (a+b)z}$.

Again, this identity has a simple combinatorial interpretation when the numbers a and b are positive integers. In this case, consider two disjoint alphabets Σ and Σ' of respective cardinality a and b. The left-hand-side of this equality is the generating function of the shuffle of the languages $\Sigma^\star$ and $\Sigma'^\star$ and the right-hand-side is the generating function of the language $(\Sigma \cup \Sigma')^\star$, both languages being trivially equal. The proof in the general case is computational.

Proof of Lemma 5. We have by definition

$$\frac{1}{1 - az} \circledcirc \frac{1}{1 - bz} = \sum_{n \ge 0} \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} z^n = \sum_{n \ge 0} (a+b)^n z^n = \frac{1}{1 - (a+b)z}.$$

Note that, in particular, $\frac{1}{1-az} \circledcirc \frac{1}{1+az} = 1$. $\square$

Using the identities presented above, one can compute the coloured product of any two poles as follows:

$$\begin{aligned} \left(\frac{1}{1 - az} \right)^k \circledcirc \left(\frac{1}{1 - bz} \right)^\ell &= \left((1 + az)^{k-1} \circledcirc (1 + bz)^{\ell-1} \right) \circledcirc \frac{1}{1 - (a+b)z} \\ &= \sum_{j=1}^{k+\ell-1} \frac{\lambda_{k,\ell,j}(a,b)}{(1 - (a+b)z)^j} \end{aligned} \qquad (1.14)$$

where the $\lambda_{k,\ell,j}(a,b)$ are computable coefficients. In particular, the coefficient of highest degree $\lambda_{k,\ell,k+\ell-1}(a,b)$ is given by $\frac{a^{k-1} b^{\ell-1}}{(a+b)^{k+\ell-2}} \binom{k+\ell-2}{k-1}$.

More generally, using the partial fraction decomposition of two rational functions, one can compute their coloured product using (1.14). This shows that rational functions are stable by coloured product, thus proving that the generating function of the executions and the generating function of the prefixes of a program are rational.

In the particular case where both functions are generating functions, this also gives us that, if their respective dominant singularities are ρ_1 and ρ_2 of degrees α_1 and α_2, then the dominant singularity of their coloured product is $(\rho_1^{-1} + \rho_2^{-1})^{-1}$ and is of degree $\alpha_1 + \alpha_2 - 1$. We will use this fact in the proof of Theorem 11.

Proof of Theorem 11. We prove by induction that, if P is an NFJ program with an infinite state-space, then its generating function of executions $P(z)$ and its generating function of prefixes $\bar{P}(z)$ have the same radius of convergence ρ and satisfy $\bar{P}(z) \sim \lambda P(z)$, for some constant $\lambda > 0$, when $z \longrightarrow \rho$. Recall that $P(z)$ and $\bar{P}(z)$ are rational.

Parallel composition. If $P = (Q \parallel R)$ and P has an infinite state-space, then at least one of Q or R has an infinite state-space too. Assume, without loss of generality, that this is the case for Q. Then, by induction hypothesis, $Q(z)$ and $\bar{Q}(z)$ have the same radius of convergence ρ_1 and we have $\bar{Q}(z) \sim \lambda_1 Q(z) \sim \frac{\bar{C}_1}{(1-z/\rho_1)^{\alpha_1}}$, for some positive constants λ_1, $\bar{C}_1$ and α_1 when $z \longrightarrow \rho_1$.

- If R has a finite state-space, it is easy to see that $R(z)$ and $\bar{R}(z)$ are polynomials and that they have the same degree k. Then, using Lemma 3, one can prove that P has the same poles as Q with the same degrees increased by k. Thus, the radius of convergence of $P(z)$ and $\bar{P}(z)$ is ρ_1 and near ρ_1 we have $\bar{P}(z) \sim \lambda P(z) \sim \frac{\bar{C}}{(1-z/\rho_1)^{\alpha_1+k}}$ for some positive constants λ and $\bar{C}$.
- If R has an infinite state-space too, then by induction hypothesis $R(z)$ and $\bar{R}(z)$ have the same radius of convergence ρ_2 and we have near ρ_2 $\bar{R}(z) \sim \lambda_2 R(z) \sim \frac{\bar{C}_2}{(1-z/\rho_2)^{\alpha_2}}$ for some positive constants λ_2, $\bar{C}_2$ and α_2. By using Lemma 5 and Lemma 4 to compute the partial fraction decomposition of $P(z)$ and $\bar{P}(z)$, we can show that they have the same radius of convergence $\rho = (\rho_1^{-1} + \rho_2^{-1})^{-1}$ and that near ρ we have $\bar{P}(z) = \lambda_1\lambda_2 P(z) \sim \frac{\bar{C}}{(1-z/\rho)^{\alpha_1+\alpha_2-1}}$ for some computable constant $\bar{C}$.

Non-deterministic choice. If $P = (Q + R)$ and P has an infinite state-space, then at least one of Q or R has an infinite state-space too. Similarly as before, we assume without loss of generality that this is the case for Q and we apply the induction hypothesis to Q with the same notations.

- First consider the case where either R has a finite state-space, or R has an infinite state-space but with $\rho_2 > \rho_1$ or with $\rho_1 = \rho_2$ and $\alpha_2 < \alpha_1$. In this case of the radius of convergence of $\bar{P}(z)$ and $P(z)$ is ρ_1 and near ρ_1 we have $\bar{P}(z) \sim \bar{Q}(z) \sim \lambda Q(z) \sim \lambda P(z)$ since $R(z) = o(Q(z))$.
- The symmetric case is similar by commutativity, so it only remains to handle the case where $\rho_1 = \rho_2$ and $\alpha_1 = \alpha_2$. In this case we have $\bar{P}(z) \sim \frac{\bar{C}_1+\bar{C}_2}{(1-z/\rho_1)^{\alpha_1}}$ and $P(z) \sim \frac{\lambda_1^{-1}\bar{C}_1+\lambda_2^{-1}\bar{C}_2}{(1-z/\rho_1)^{\alpha_1}}$, which allows to conclude.

Sequential composition. If $P = (Q; R)$ and P has an infinite state-space, then at least one of Q and R have an infinite state-space too. We have that $\bar{P}(z) = \bar{Q}(z) + Q(z)(\bar{R}(z) - 1)$ and $P(z) = Q(z)R(z)$. Again, we use the same notations as above.

- We first consider the case where Q has an infinite state-space and either the state-space of R is finite or is infinite but is such that $\rho_2 > \rho_1$. Thus, the dominant singularity of $P(z)$ and $\bar{P}(z)$ is ρ_1 and when $z \longrightarrow \rho_1$ we have $\bar{P}(z) \sim (\lambda_1 + \bar{R}(\rho_1) - 1)Q(z)$ and $P(z) \sim R(\rho_1)Q(z)$. As a consequence $\bar{P}(z) \sim \frac{\lambda_1 + \bar{R}(\rho_1) - 1}{R(\rho_1)} P(z)$.
- In the symmetric case, that is either the state-space of Q is finite or is infinite but such that $\rho_1 > \rho_2$, we have that the radius of convergence of $\bar{P}(z)$ and $P(z)$ is ρ_2. Besides, near ρ_2 we have $\bar{P}(z) \sim Q(\rho_2)\bar{R}(z) \sim Q(\rho_2)\lambda_2 R(z)$ and $P(z) \sim Q(\rho_2)R(z)$. Thus $\bar{P}(z) \sim \lambda_2 P(z)$.
- Finally, if both Q and R have an infinite state-space and $\rho_1 = \rho_2$, then the radius of convergence of $P(z)$ and $\bar{P}(z)$ is ρ_1 and near ρ_1 we have $\bar{P}(z) \sim Q(z)\bar{R}(z) \sim Q(z)\lambda_2 R(z) \sim \frac{\lambda_1^{-1}\bar{C}_2\bar{C}_1}{(1-z/\rho_1)^{\alpha_1+\alpha_2}}$ and $P(z) \sim Q(z)R(z)$, which allows to conclude.

Loops. If $P = Q^\star$, then we have $P(z) = (1 - (Q(z) - Q(0)))^{-1}$ and there is a unique $\rho > 0$ such that $Q(\rho) - Q(0) = 1$. If Q has an infinite state-space, then we have that ρ is smaller than the radius of convergence of $Q(z)$ (and $\bar{Q}(z)$, by induction hypothesis). In a neighbourhood of ρ we have $Q(z) - Q(0) = 1 - (\rho - z)Q'(\rho) + o(\rho - z)^2$, with $Q'(\rho) > 0$, and thus $P(z) \sim \frac{Q'(\rho)^{-1}}{\rho - z}$. Finally, observe that $\bar{P}(z) = 1 + P(z)(\bar{Q}(z) - 1) \sim P(z)(\bar{Q}(\rho) - 1)$ near ρ.

Base cases. There is nothing to prove for $P = 0$ or $P = a$. Informally, the "real" base cases of this induction, that is the cases where P has an infinite state-space but all its sub-terms have a finite one, is the case where $P = Q^\star$ and $Q \neq 0$ contains no loop. □

1.4.3 Uniform random sampling of prefixes

We now tackle the problem of sampling a uniform prefix of a given length n, of a given program. We first describe in Algorithm 12 how to compute the generating functions of the prefixes of all the sub-terms of a given NFJ program recursively. As for the generating function of the executions, the algorithm must store each resulting generating function in its corresponding AST node. Moreover, we assume that Algorithm 8 has already been called on a program P before running Algorithm 8 on it, so that the generating functions of the *executions* of the necessary sub-terms of P are available.

Algorithm 12 Computation of the generating function of the prefixes of a program up to degree n.

```
function prefGfun(P, n)
    if P = 0 then return 1
    else if P = a then return 1 + z
    else if P = Q + R then return prefGfun(Q, n) + prefGfun(R, n) − 1
    else if P = Q ∥ R then return prefGfun(Q, n) ⊚ prefGfun(R, n)
    else if P = Q; R then
        q(z) ⟵ gfun(Q, n)                          ▹ should have been pre-computed
        return prefGfun(Q, n) + q(z) · (prefGfun(R, n) − 1)
    else if P = Q⋆ then
        p(z) ⟵ gfun(P, n)                          ▹ should have been pre-computed
        return 1 + p(z) · (prefGfun(Q, n) − 1)
```

Random sampling is then straightforward, the methodology is the same as in the previous section. Algorithm 13 describes a uniform random sampler of prefixes based on the generating functions of prefixes computed in Algorithm 12.

Algorithm 13 Uniform random sampling of prefixes of a given length

Input: An NFJ program P and a length n
Output: A uniform prefix of execution of P of length n

```
function UnifPref(P, n)
    if n = 0 then return the empty prefix
    else if P = a then return a
    else if P = Q + R then
        if Bernoulli(q̄_n / (q̄_n + r̄_n)) then return UnifPref(Q, n)
        else return UnifPref(R, n)
    else if P = Q ∥ R then
        draw k ∈ ⟦0; n⟧ with probability (n choose k) q̄_k r̄_{n−k} / p̄_n
        return shuffle(UnifPref(Q, k), UnifPref(R, n − k))
    else if P = Q; R then
        if Bernoulli(q̄_n / p̄_n) then return UnifPref(Q, n)
        else
            draw k ∈ ⟦0; n⟧ with probability q_k r̄_{n−k} / (p̄_n − q̄_n)
            return concat(UnifExec(Q, k), UnifPref(R, n − k))
    else if P = Q⋆ then
        draw k ∈ ⟦0; n − 1⟧ with probability p_k q̄_{n−k} / p̄_n
        return concat(UnifExec(P, k), UnifPref(Q, n − k))
```

The complexity analysis of Algorithm 13 is the same as that of Algorithm 7 in the previous section, and arrives to the same conclusion. We do not repeat it here.

1.4.4 Experimental study

Performance evaluation

In order to assess experimentally the efficiency of our method, we put into use the algorithms presented here and demonstrate that they can handle systems with a significantly large state space. We generated a few NFJ programs as described in Section 1.3.4 and conducted a similar experiment on the same machine. We quickly recall the main points of our setup below.

Table 1.9 on the following page reports the runtime of the preprocessing phase (Algorithm 12), the runtime of the random sampler (Algorithm 13) and the number of prefixes of length n for various programs and various values of n. Here again, for the runtime of the counting algorithm, every measurement was performed 7 times and we reported the median of these 7 values. For the random sampler, every measure was performed 101 times and for each one we report the median of these values as well as the interquartile range (IQR)[8], which gives an idea of the dis-

[8]The interquartile range of a set of measures is the difference between the third and the first quartiles. Compared with the value of the median, it gives a rough estimate of the dispersion of the measures.

Table 1.9: Quick benchmark of the counting and random sampling functions of execution prefixes

$\lvert P \rvert_c$	n	# prefixes	mem. size	prefGfun	UnifPrefix	IQR
100	500	$1.841 \cdot 2^{1128}$	1.77M	0.025s	0.250ms	0.013ms
100	1000	$1.123 \cdot 2^{2244}$	6.67M	0.087s	0.635ms	0.662ms
500	500	$1.640 \cdot 2^{1611}$	5.75M	0.173s	0.372ms	0.024ms
500	1000	$1.034 \cdot 2^{3124}$	20.90M	1.002s	1.098ms	0.069ms
1000	500	$1.047 \cdot 2^{2462}$	17.84M	0.563s	0.526ms	0.041ms
1000	1000	$1.523 \cdot 2^{4844}$	67.14M	3.223s	1.962ms	0.191ms
2000	500	$1.685 \cdot 2^{2381}$	21.43M	0.673s	0.475ms	0.047ms
2000	1000	$1.098 \cdot 2^{4732}$	79.98M	3.630s	1.155ms	0.038ms

persion of the measures. We use these metrics rather than the mean and the variance to reduce the importance of extreme values and give a precise idea of what runtime the user should expect when running our sampler. The time reported is the CPU time as measured by C's `clock` function. The state-space column indicates the number of *prefixes* of length n. The mem. size column reports the amount of memory occupied by the generating functions of the executions *and* that of the executions prefixes. Recall that both generating functions are necessary for the random sampling routine.

The take-away of this experiment is that (1) the preprocessing phase can be carried out for systems with a state-space of size $\approx 2^{18000}$ in a time of the order of the minute and that (2) once this is done, sampling a uniform prefix in this set is a matter of a few milliseconds.

Prefix covering

We present another experimentation here that highlights the importance of the uniform distribution for the purpose of state-space exploration. The problem is the following. We consider given an NFJ program and we sample random prefixes of a given length n of this program using two different algorithms:

- our random sampler which is *globally* uniform among all prefixes of length n;
- a more "naive" sampler that repeatedly generates one execution step uniformly at random among the legal steps, until we get a length n prefix. This strategy is called *locally uniform* or *isotropic*.

The question is: in average, how many random prefixes must be generated in order to discover a given proportion of the possible prefixes? This question actually falls under the scope of the Coupon Collector Problem, which is treated in depth in [FGT92]. Table 1.10 on the next page gives numerical answers for both exploration strategies for a random NFJ program of size 25 and for a target coverage of 20% of the possible prefixes.

Expectedly the uniform strategy is faster but what is interesting to see is that the speed-up compared to the isotropic method grows extremely fast. The more the state-space grows, the more the uniform approach is unavoidable.

Unfortunately, the formula given in [FGT92] for the isotropic case involves the costly computation of power-sets which makes it impractical to give values for larger programs and prefix

Table 1.10: Expected number of prefixes to be sampled to discover 20% of the prefixes of a random program of size 25 with either the isotropic or the uniform method

Prefix length	1	2	3	4	5
# prefixes	11	18	30	60	128
Isotropic	2.1	4.45	11.17	35.09	$1.28 \cdot 10^{14}$
Uniform	2.1	3.18	6.57	13.26	27.69
Gain	0%	40%	70%	165%	$4.61 \cdot 10^{14}$%

lengths. However, these small-size results already establish a clear difference between the two methods. It would be interesting to have theoretical bounds to quantify this explosion or to investigate more efficient ways to compute these values but this falls out of the scope of this work.

Chapter 2

Directed Ordered Acyclic Graphs and Multi-Graphs

In this chapter, we dive into a second aspect of this thesis: the study of some classes of directed acyclic graphs (DAGs). This work is motivated by the use of DAGs as a simpler encoding of the control flow of programs than partial orders. Some of the results presented here have been published in the paper [GPV21][1].

The outline of this chapter is as follows. In Section 2.2 we introduce the model of Directed Ordered Acyclic Graph (DOAG) and describe a recursive decomposition which is amenable to efficient enumeration and recursive random sampling. In Section 2.3, we show that the same decomposition scheme is applicable to the classical model of labelled DAGs and we obtain new recurrence formulas for counting them. This new decomposition of labelled DAGs yields a straightforward recursive sampler of labelled DAGs with a given number of vertices, edges and sources, which is a new result. Finally, in Section 2.4, we study a natural extension of DOAGs which consists in allowing multiple edges to exist between two vertices.

2.1 Context and related work

DAGs usually come in two different flavours: labelled and unlabelled. In a labelled DAG, the vertices are all distinguishable and usually identified by an integer. Labelled DAGs are thus adequate to encode a set of relations between a set of known elements (the vertices). On the other hand, unlabelled DAGs are defined as labelled DAGs taken up to a relabelling, that is an edge-preserving permutation of the vertices. They thus "forget" the distinction between the vertices and only retain the structure of the graph.

Counting The first results on *labelled* DAG enumeration were obtained by Robinson [Rob73] and Stanley [Sta73] in the 1970s using two different approaches. Stanley obtained his counting formula by manipulations on generating functions and on the chromatic polynomials while

[1][GPV21] "Unlabelled ordered DAGs and labelled DAGs: constructive enumeration and uniform random sampling" has been accepted for publication in the proceedings of the LAGOS conference in 2021.

Robinson described a recursive “layer-by-layer” decomposition of DAGs. Robinson’s decomposition consists in removing all the sources of a DAG at once and counting the number of possible outcomes of this operation if the initial DAG had k sources and n vertices. This is pictured in Figure 2.1. The case of *unlabelled* DAGs was also solved by Robinson in [Rob77] using the same decomposition and resorting to Burnside’s lemma and the inclusion-exclusion principle to take the symmetries into account. In all three articles DAGs are counted by *number of vertices* and the number of edges of the graphs is not taken into account. In the 1990s Gessel [Ges95; Ges96] used a similar kind of generating functions as Stanley, called *graphic generating functions*, to provide a more precise enumeration including more parameters. In particular [Ges96] counts labelled DAGs by sources, sinks, vertices, and edges. Gessel’s approach also relies on Robinson’s “layer-by-layer” decomposition so that [Sta73] is the only notable exception where the problem is tackled from a different angle. In this chapter we take a slightly different approach too and decompose the graphs by removing *only one source* at a time. Although this is a minor difference, this leads to new recurrence formulas with important applications in terms of random generation.

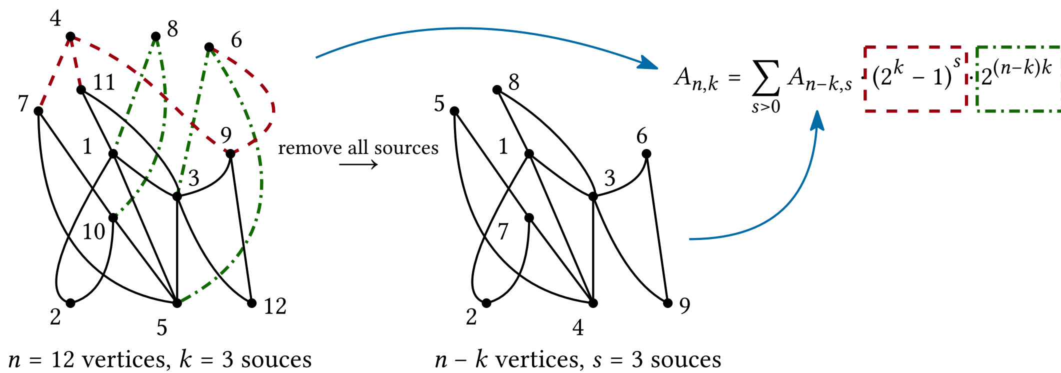


Figure 2.1: Robinson’s layer-by-layer decomposition of labelled DAGs. Edges are implicitly oriented from top to bottom. In order to make a DAG with n vertices, including k “top” sources, out of a smaller DAG with $(n - k)$ vertices, including s “lower” sources, one needs to connect each of the s lowers sources to at least one top source, hence the $(2^k - 1)^s$ term. In addition, there might be an edge from any the of k top sources to any of the $(n - k)$ other vertices.

Random generation The recurrence formula obtained by Robinson for the labelled case being based on a recursive decomposition, it naturally leads to a random sampler by the recursive method. This was first noted by [MDB01] but the authors of the papers assumed this approach to be inefficient. Instead, they developed an alternative approach based on a Markov chain which adds and removes edges at random. Later [KM15] made progress on the two methods. First they proved that the recursive approach is actually efficient, especially in comparison with the Markov chain from [MDB01]. And second, they described a second Markov chain with a faster convergence rate. A way of bounding the number of edges of the generated DAGs is discussed in [MDB01]. However, a common limitation of both approaches is that it only offers little control over this number of edges. On one hand, the recursive method only samples uniform DAGs

with n vertices, which yields dense graphs with about $\frac{n^2}{4}$ edges in average. On the other hand, the Markov chain approach offers more control as it can be constrained to sample DAGs with a given maximum number of edges. However, no study of the speed of convergence of such a chain is provided.

A second limitation of the above approaches is that it only allows to sample *labelled* DAGs. Indeed, for the purpose of approximating partial orders and, ultimately, generating programs, unlabelled DAGs are more appropriate since the goal is to generate different program *structures*. For instance, the two unlabelled DAGs with 4 vertices pictured below correspond respectively 12 and 24 distinct labelled DAGs. As a consequence, uniformly generating labelled DAGs rather than unlabelled ones introduces a bias towards structures with fewer symmetries. In our example the DAG on the right has twice as much chance of being drawn.

12 = 4!/2 distinct labellings 24 = 4! distinct labellings

The fact that unlabelled DAGs have symmetries is actually what makes them difficult to handle. Having a labelling of the vertices can be seen as a pragmatic way to *break* those symmetries so that we have some algorithms we can use in practice. In the present chapter we consider an alternative way to break symmetries by introducing an *ordering* of the outgoing edges.

In a different line of work, [FN13] describes a uniform random sampler of acyclic deterministic automata. These objects have a similar structure to the directed ordered acyclic graphs (DOAG) we introduce in this chapter, in the sense that the labels on the transitions of the automata induce an order on the outgoing edges of each state. However, DOAGs form a strictly larger class so that the algorithm developed there does not directly apply. The approach used in [FN13] also follows the layer-by-layer decomposition schemes and is based on the recursive method.

Asymptotic results More recently, asymptotic results were obtained for the number of optimally compacted binary trees in [EFW21]. Compacted trees are trees where common substructures are shared so that they are naturally in correspondence with DAGs. Moreover, these trees are *plane*: the order of their children matters. As a consequence, these objects form a subclass of the directed ordered acyclic graphs we consider in this chapter. Although our primary focus is on random generation rather than asymptotic enumeration, the novel techniques developed in the article are a candidate of choice to tackle some problem left open in Section 2.4.

In another line of work, analytic methods have been recently developed to tackle asymptotic enumeration problems related to directed graphs (see [PD19] and the pre-prints [PD20; Pan+20] for instance). In particular these articles enrich the symbolic method of analytic combinatorics with a new construction adapted to digraph enumeration. Their approach shows promising results and, although it does not seem to apply to DOAGs at the moment, it is a source of inspiration for future developments.

2.2 Directed Ordered Acyclic Graphs

We now present the first contribution of this chapter by introducing and studying the class of directed ordered acyclic graphs (DOAG). In Section 2.2.1 we present the class of DOAGs and describe a "vertex-by-vertex" decomposition scheme allowing us to obtain recursive formulas for counting these objects by vertices and edges. Then, in Section 2.2.3 we cover some computational aspects of their enumeration and describe a uniform random sampler of DOAGs with a given number of vertices and edges based on the recursive decomposition.

Finally, as an extension, we provide a more efficient counting formula and a more efficient random sampler for DOAGs with n vertices and an arbitrary number of edges. In Section 2.2.4 we present an encoding of DOAGs as integer matrices and exhibit asymptotic bounds for the number of DOAGs of size n, based on this encoding. Then, using a combinatorial interpretation of these bounds, in Section 2.2.5 we describe an efficient uniform random sampler for these objects which does not require the costly pre-computation of their counting sequence.

An implementation of the counting algorithms and the recursive random samplers presented in this section is available on Github as a C library at the address `https://github.com/Kerl13/randdag`.

2.2.1 Definition and recursive decomposition

We introduce a model of directed acyclic graphs called "Directed Ordered Acyclic Graphs" (or DOAGs) which is similar to the classical model of unlabelled DAGs but where, in addition, we have a total order on the outgoing edges of each vertex.

Definition 12 (DOAG). *A directed ordered graph is a triple $(V, E, (\prec_v)_{v \in V \cup \{\varnothing\}})$ where:*

- *V is a finite set of vertices;*
- *$E \subset V \times V$ is a set of edges;*
- *for all $v \in V$, $\prec_v$ is a total order over the set of* outgoing *edges of v;*
- *and $\prec_\varnothing$ is a total order over the set of* sources *of the graph, that is the vertices without any incoming edge.*

Two such graphs are considered to be equal *if there exists a bijection between their respective sets of vertices that preserves both the edges and the order relations $\prec_v$ and $\prec_\varnothing$. Finally, a DOAG is a directed ordered graph $(V, E, (\prec_v)_{v \in V \cup \{\varnothing\}})$ with only one sink and such that (V, E), as a directed graph, is acyclic.*

The restriction we put on the number of sinks is a way to ensure the weak connectivity of the graph. Moreover, although we study this class as a whole, we believe that the DOAGs with only one source are of higher interest, in particular for the purpose of modelling programs which generally have a single starting point. As an example, all the DOAGs with one source and up to 4 edges are pictured in Fig. 2.2.

We describe a canonical way to recursively decompose a DOAG into smaller structures. The idea is to remove vertices one by one in a deterministic order, starting from the smallest source (with respect to their ordering $\prec_\varnothing$). Formally, we define a decomposition step as a bijection between the set of DOAGs with at least two vertices and the set of DOAGs given with some extra information. Let D be a DOAG with at least 2 vertices and consider the new graph D'

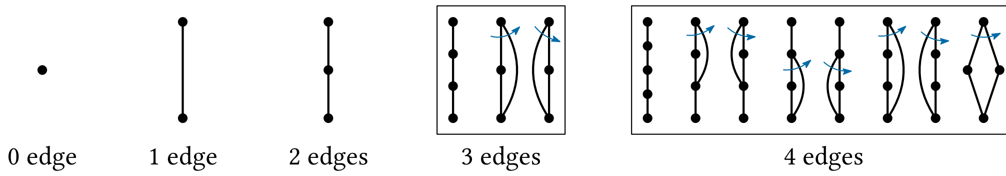


Figure 2.2: All DOAGs with one source and up to 4 edges. All edges are implicitly oriented from top to bottom and the order of the outgoing edges of each vertex is indicated by the thinner blue arrows (always from left to right here).

obtained from D by removing its *smallest* source v and its outgoing edges. We also need to specify the ordering of the sources of D'. We consider the ordering where the *new* sources of D' (those that have been uncovered by removing v) are considered to be in the *same order* (with respect to each other) as they appear as children of v and all *larger* than the other sources. The additional information necessary to reconstruct D from D' is the following:

1. the number s of sources of D' which have been uncovered by removing v;
2. the set I of internal (non-sources) vertices of D' such that there was an edge in D from v to them;
3. the function $f : I \longrightarrow [\![1; s+|I|]\!]$ identifying the positions, in the list of outgoing edges of v, of the edges pointing to an element of I.

In fact, this decomposition describes a bijection between DOAGs with at least 2 vertices and quadruples (D', s, I, f) where D' is a DOAG with k' sources, I is a subset of its internal vertices, $0 \leq s \leq k'$ is a non-negative integer, $s + |I| > 0$ and $f : I \longrightarrow [\![1; s + |I|]\!]$ is an injective function. Indeed, the inverse transformation is as follows. Create a new source v with $s + |I|$ outgoing edges such that the i-th of these edges is connected to $f^{-1}(i)$ when $i \in f(I)$ and is connected to one of the s largest sources of D' otherwise. The s largest sources of D' must be connected to the new source exactly once and in the same order as they appear in the list of sources of D'. Note that the order in which the vertices are removed when iterating this process corresponds to a BFS-based topological sort of the graph. Fig. 2.3 pictures the first 3 decomposition steps of an example DOAG.

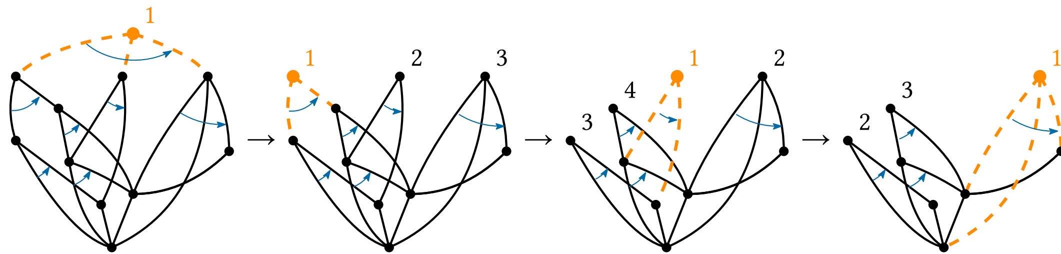


Figure 2.3: Recursive decomposition of a DOAG by removing sources one by one in a breadth first search (BFS) fashion. The edges are implicitly oriented from top to bottom and the order of the outgoing edges of each vertex is indicated by the thinner blue arrows (always from left to right here). The integer labels at each stage indicate the ordering of the sources.

This decomposition can be used to establish a recursive formula for counting DOAGs, which

is given below. Let $D_{n,m,k}$ denote the number of DOAGs with n vertices, m edges and k sources, then we have:

$$\begin{aligned} D_{1,m,k} &= \mathbb{1}_{\{m=0 \wedge k=1\}} \\ D_{n,m,k} &= 0 && \text{when } k \leq 0 \\ D_{n,m,k} &= \sum_{p=1}^{\min(n-k,m+2-n)} \sum_{i=0}^{p} D_{n-1,m-p,k-1+p-i} \binom{n-k-p+i}{i} \binom{p}{i} i! && \text{otherwise,} \end{aligned} \tag{2.1}$$

where $p = s + i$ corresponds the out-degree of the smallest source, the term $\binom{n-k-p+i}{i} = \binom{n-k-s}{i}$ accounts for the choice of the set I and the term $\binom{p}{i} i!$ accounts for the number of injective functions $f : I \longrightarrow [\![1; p]\!]$. The bounds on p in the sum is justified by two combinatorial arguments. First, since p is the out-degree of the smallest source, it cannot be zero and it is upper bounded by the number of vertices it might have an outgoing edge to, that is the number $(n-k)$ of non-source vertices of the graph. The second upper bound is obtained by observing that all vertices but the sink have at least one outgoing edge, hence the total number of edges m is at least $p + (n - 2)$.

Remark 2. *Since $p = i + s$ is the out-degree of the removed source, the sequence $D^{(d)}_{n,m,k}$ counting the DOAGs of maximum out-degree bounded by a given constant d can easily be obtained by replacing the bound over p by $\min(n - k, m + 2 - n, d)$ in the outermost sum.*

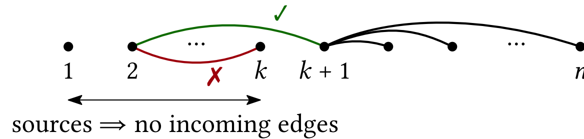
2.2.2 Computational aspects of the enumeration

We consider the problem of computing $D_{n,m,k}$ for all n, m and k up to a given bound. This can be achieved easily using equation (2.1) and a dynamic programming approach. We do not give the algorithm here as it is a straightforward implementation of the above formula. But in this section we give some details on its computational aspects. First, in Lemma 6 we characterise the indices n, m, k such that $D_{n,m,k} > 0$. This can be used to avoid unnecessary recursive calls and to choose a memory-efficient data-structure for storing the results. Then in Theorem 12 we give the complexity of the counting procedure in terms of bitwise operations.

Lemma 6. *For $n > 1$, we have $D_{n,m,k} \neq 0$ if and only if $1 \leq k < n$ and $n - 1 \leq m \leq \binom{n}{2} - \binom{k}{2}$.*

Proof. There is always at least one source in a DOAG. Furthermore, since $n > 1$, the unique sink cannot be a source, thus $1 \leq k < n$ is a necessary condition for $D_{n,m,k}$ to be positive.

Now let n and k be such that $1 \leq k < n$ and consider n vertices labelled from 1 to n. The maximum possible of edges in a DOAGs with k sources is obtained by putting an edge from vertex i to vertex j if and only if $i < j$ and $j > k$ as pictured below.



This corresponds to $\binom{n}{2} - \binom{k}{2}$ edges since there are $\binom{n}{2}$ pairs (i, j) such that $i < j$ and $\binom{k}{2}$ pairs (i, j) such that $i < j \leq k$. Furthermore, it is possible to remove any number of edges from this maximal

case while keeping the DOAG weakly connected as long as the total number of edges remains greater or equal to $(n-1)$. If $m < n-1$, the graph is necessarily disconnected. □

In Theorem 12 we get a straightforward upper bound on the number of arithmetic operations necessary to compute all the $D_{n,m,k}$ up to certain bounds. As it is usual in combinatorial enumeration, there is a hidden cost factor in the *size* of the numbers at stake: the more they grow the more costly arithmetic operations become. To account for this cost we also give an upper bound on the bit-size of all numbers being multiplied.

Theorem 12. *Let $N, M > 0$ be two integers. Computing $D_{n,m,k}$ for all $n \le N$, $m \le M$ and all possible k can be done with $O(N^4 M)$ multiplications of integers of size at most $O(M \ln M)$.*

The first part of Theorem 12 is straightforward but we need a bound on the value of $D_{n,m,k}$ for the second part, which is the purpose of Lemma 7.

Lemma 7. *For all n, m, k, we have* $D_{n,m,k} \le \binom{\binom{n}{2} - \binom{k}{2}}{m} \cdot (m-n+2)!$

Proof. This upper bound is based on two combinatorial arguments. Consider a sequence of n vertices obtained by decomposing a DOAG D with k sources and m edges. The first k vertices of this sequence thus correspond to the k sources of D.

First, the set of edges of D is a subset of size m of the set of all pairs of vertices that are not made of two sources. Note that not all such subsets form a valid DAG however. Hence, the number of ways to choose the m edges of D is bounded above by $\binom{\binom{n}{2}-\binom{k}{2}}{m}$. Second, the number of ways to order the outgoing edges of all the vertices is bounded by $d_1!d_2!\cdots d_n!$ where d_j denotes the out-degree of the i-th vertex. Finally, this product is bounded by $(m-n+2)!$, which corresponds to the case where all the d_j with $i < n$ but one are equal to 1, $d_n = 0$ and the remaining one is equal to $(m-n+2)$. □

This bound on the number of DOAGs is rough but it is precise enough to get an estimation of the bit-size of these numbers.

Corollary 2. *There exists a constant $c > 0$ such that for all n, m, k we have $\log_2(D_{n,m,k}) \le c \cdot m \cdot \log_2 m$.*

Proof. Let $L = \binom{n}{2} - \binom{k}{2}$. By Lemma 7, we have that:

$$D_{n,m,k} \le \frac{L(L-1)(L-2)\cdots(L-m+1)}{m!} \cdot (m-n+2)! \le L(L-1)(L-2)\cdots(L-m+1) \le L^m.$$

Hence, $\log_2(D_{n,m,k}) \le m\log_2(L)$. Moreover $\log_2(L) \le \log_2(n^2) = 2\log_2(n)$ and since $n \le m+1$ we have $\log_2(n) = O(\log_2(m))$. □

We now have enough information to prove Theorem 12.

Proof of Theorem 12. Since $D_{n,m,k} = 0$ for $k > n$, we need to compute $O(N^2 M)$ numbers. Moreover, computing each $D_{n,m,k}$ requires to compute a sum of at most n^2 terms, each of which is the product of a number of bit-length $O(m \ln m)$ with a coefficient of the form $C(j,p,i) = \binom{j+i}{i}\binom{p}{i} i!$ (for some $j, p, i \leq n$) of bit-length $O(n \ln n)$. Overall this accounts for $O(N^4 M)$ multiplications of bit-complexity $\mathcal{M}(M \ln M)$.

There remains to measure the cost of computing the coefficients $C(j,p,i)$. They can be obtained at a small amortized cost using the relation $C(j,p,i) = C(j,p,i-1) \cdot \frac{(j+i)(p-i+1)}{i}$ (for all $1 < i \leq p$) to get the value of the coefficient at i from its value at $i-1$ when summing the terms of (2.1) for increasing values of i. Clearly, the cost of multiplying numbers of bit-length $n \ln n$ and $\ln n$ is bounded by $\mathcal{M}(n \ln n)$ and therefore $\mathcal{M}(M \ln M)$ since $n \leq M + 1$. □

2.2.3 Counting and sampling algorithms

In this section we describe a uniform random sampler of DOAGs based on the recursive decomposition given in the previous section. Our algorithm is based on the so-called "recursive method" from [NW78] in the way we select the parameters of the sub-structures. However, unlike what we would expect in the systematised framework from [FZV94], the substructures are not independent. Once the sub-DOAG D' accounted for by $D_{n-1,m-p,k-1+p-i}$ has been selected, the set I and the injective function $f : I \longrightarrow [\![1; |I| + s]\!]$ accounted for by $\binom{n-k-p+i}{i}\binom{p}{i} i!$ cannot be sampled independently from D'.

Our random sampler is given in Algorithm 14. We first give a high-level description of the algorithm here for the sake of readability. Implementation considerations are discussed below, and in particular in Algorithm 15 we give a fast algorithm for the generation of the outgoing edges of the new source at each step of the global random sampling procedure.

Algorithm 14 Recursive uniform sampler of DOAGs

Input: Three integers (n, m, k) such that $D_{n,m,k} > 0$
Output: A uniform random DOAG with n vertices (including k sources) and m edges

1: **function** UnifDOAG(n, m, k)
2:     **if** $n \leq 2$ **then** generate the (unique) DOAG with n vertices
3:     **else**
4:         **pick** (p, i) with probability $D_{n-1,m-p,k-1+p-i}\binom{n-k-p+i}{i}\binom{p}{i} i!/D_{n,m,k}$
5:         $D' \longleftarrow$ UnifDOAG$(n-1, m-p, k-1+p-i)$
6:         $I \longleftarrow$ a uniform subset of size i of the inner vertices of D'
7:         $I' \longleftarrow$ a uniform permutation of I
8:         $E \longleftarrow$ a uniform shuffling of I' with the $s = p - i$ largest sources of D'
9:         **return** the DOAG obtained by adding a new source to D' with E as its list of outgoing edges

The **pick** instruction at line 4 implements the "recursive method" scheme: pick the parameters of the sub-structures using the pre-computed counting information. One possible way to implement this is to draw a random variable $r \longleftarrow$ Unif$([\![0; D_{n,m,k} - 1]\!])$ and to compute the partial sum of the terms $D_{n-1,m-p,k-1+p-i}\binom{n-k-p+i}{i}\binom{p}{i} i!$ (in any order independent of r) until the sum is

greater than r. The indices p and i of the last term of the sum are the ones to pick. Independently of the summation order, this procedure has complexity $O(n^2)$ in the worst case in terms of multiplications of big integers. An interesting order of summation is the one where the pairs (p, i) are taken in lexicographic order. In this case the complexity of the **pick** function can be expressed as $O(p^2)$ which is more informative about the cost of sampling the whole DOAG since p is the *out-degree* of the new source. We consider that this order is used here.

Once we have sampled the sub-DOAG D', sampling I is straightforward and the injective function f is obtained as a permutation of S (thus deciding of the order of the elements of I as children of the new vertex) shuffled with the largest $(p - i)$ sources of D'. The correctness and complexity of this procedure in terms of integer multiplications are stated in Theorem 13.

Theorem 13. *Algorithm 14 computes a uniform random DOAG of given parameters n, m and k by performing $O\left(\sum_v d_v^2\right)$ multiplications where v ranges over the vertices of the resulting graph and d_v is the out-degree of v.*

Proof. The complexity result is a straightforward consequence of the above discussion. Uniformity is proven by induction. Let D be a DOAG of parameters (n, m, k) and let (D', s, I, f) denote the result of one decomposition step of D. Then the probability that D is returned by UnifDOAG(n, m, k) is

$$\mathbb{P}[D'] = \mathbb{P}[(p, i)] \cdot \mathbb{P}[D'|p, i] \cdot \mathbb{P}[I|D', i] \cdot \mathbb{P}[f|I, p, i]$$

where, by induction we have $\mathbb{P}[D'|p, i] = 1/D_{n-1,m-p,k-1+p-i}$ and by definition:

$$\begin{aligned}
\mathbb{P}[(p, i)] &= D_{n-1,m-p,k-1+p-i}\binom{n-k-p+i}{i}\binom{p}{i} i!/D_{n,m,k} \\
\mathbb{P}[I|D', i] &= \binom{n-k-p+i}{i}^{-1} \\
\mathbb{P}[f|I, p, i] &= \left(i!\binom{p}{i}\right)^{-1}.
\end{aligned}$$

In the end, we get that $\mathbb{P}[D'] = 1/D_{n,m,k}$. □

Note that the sum $\sum_v d_v^2$ is of the order of m^2 in the worst case but can be significantly smaller if the out-degrees of the vertices are evenly distributed. In the best case we have $d_v \sim \frac{m}{n}$ for most of the vertices and as a consequence $\sum_v d_v^2 \sim m^2/n$.

We have decomposed the generation of the new source into several steps in Algorithm 14 (lines 5 to 8) to make the role of each term in the counting formula apparent, and help stating the uniformity. However there is a faster way to implement this part of the Algorithm by sampling I and its ordering *together* using a variant of the well-known Fisher–Yates algorithm (see [FY48]) using the property that the first i terms of a uniform permutation form a uniform ordered subset of size i of its elements. This is described in Algorithm 15 which can substitute lines 5 to 8 in Algorithm 14 in a practical implementation.

The first loop of Algorithm 15 (at line 4) implements the Fisher–Yates algorithm with an early exit after i iterations rather than ℓ_T. After this, the first i elements of T represent the set I

Algorithm 15 Optimised uniform sampler of new sources with given parameters

Input: Two non-negative integers i and s, an array S of length $\ell_S \geq s$ vertices playing the role of sources and an array T of length $\ell_T \geq i$ vertices playing the role of internal vertices.

Output: An array v of $i + s$ vertices, representing a new vertex with s out-edges to the s last elements of S (appearing in the same order as in S) and i edges to elements of T, chosen uniformly at random.

```
1: i' ← i
2: s' ← s
3: v ← new array of length (i + s)
4: for j = 0 to i − 1 do
5:     r ← Unif([[j; ℓ_T − 1]])
6:     T[j] ↔ T[r]
7: while i' + s' > 0 do
8:     if Ber(i'/(i' + s')) then
9:         v[i' + s' − 1] ← T[i' − 1]
10:        i' ← i' − 1
11:    else
12:        v[i' + s' − 1] ← S[ℓ_S − s + s' − 1]
13:        s' ← s' − 1
```

and their ordering is uniform. The second loop (at line 7) implements the shuffling of I with the last s elements of S. We populate the array v in reverse order so as to ensure that the elements coming from S remain sorted.

This algorithm achieves linear complexity in $p = (i + s)$ in terms of memory accesses and number of calls to the random number generator, but needs to modify T in place. Since T represents the internal vertices of a DOAG, this means that we must choose a data structure for DOAGs that is not sensitive to the order of its internal vertices.

The idea is to represent a DOAG with n vertices and k sources as an array of vertices where the first k elements are the sources, sorted in increasing order, and the other $n - k$ elements are the internal nodes stored in an unspecified order. Vertices are represented as *pointers* to arrays of vertices, the order of the elements encodes the order of the edges.

One can then allocate a single array of size n before the first call to the sampler and populate it from right to left in the recursive calls. The invariant is that, after each recursive call of the form UnifDOAG(n', m', k'), the n' last elements of the array represent its resulting DOAG D' of size n'. Algorithm 15 is then used by taking the $n' - k'$ last elements of the array as T and the k' elements preceding them as S, without making any copy. Finally the newly generated source is stored at index $n - n'$, just before the n' vertices representing D'. The advantage of this memory layout is that after this point, the s former sources that have been turned into internal nodes are already at the right place. The memory representation discussed above is pictured in Figure 2.4 on the following page.

Remark 3. *If the sequence $D^{(d)}_{n,m,k}$ from Remark 2 is used in place of $D_{n,m,k}$ in the algorithm, and without any further change, we obtain a uniform random sampler of DOAGs of maximum out-degree bounded by d. A large uniform random DOAG with bounded out-degree, sampled using this algorithm, is shown in Figure 2.5 on page 70.*

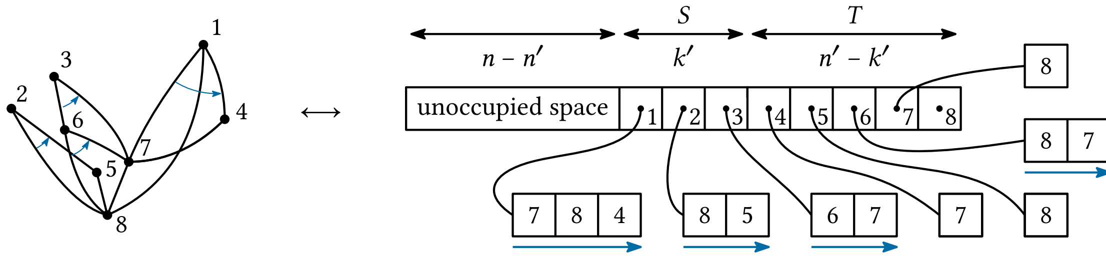


Figure 2.4: The memory layout used for storing DOAGs during the generation. The labels on the vertices of the DOAG on the left correspond to the order in which they are consumed by the decomposition. Each vertex on the left corresponds to a cell of the main array on the right. Each cell (except for the last one) has a pointer to an array of vertices representing its outgoing edges (here represented as integers for readability but stored as pointers to the corresponding array of outgoing edges in practice). For instance the cell labelled 1 has three outgoing edges to the vertices labelled 7, 8 and 4.

2.2.4 Counting DOAGs by vertices only: asymptotic results

In this section, we introduce the notion of *labelled transition matrices* to give an alternative point of view on DOAGs. Approaching the counting problem from this angle, we manage to provide lower and upper bounds on the number of DOAGs with n vertices (and any number of edges). These bounds are precise enough to give a good intuition on the asymptotic behaviour of these objects. Building on this same approach, we provide an efficient uniform sampler of DOAGs with n vertices in the next section.

The decomposition scheme described in Section 2.2.1 corresponds to a traversal of the DOAG. This allows to label its vertices from 1 to n and to consider its transition matrix using these labels as indices. Usually, the transition matrix of a directed graph D is defined as the matrix $(a_{i,j})_{1\leq i,j\leq n}$ such that $a_{i,j}$ is 1 if there is an edge from vertex i to vertex j in D, and 0 otherwise. This representation encodes the set of the edges of a DAG but not the edge ordering of DOAGs. In order to take this into account, we use a slightly different encoding.

Definition 13 (Labelled transition matrix)**.** *Let D be a DOAG with n vertices. We identify the vertices of D to the integers from 1 to n corresponding to their order in the vertex-by-vertex decomposition. The labelled transition matrix of D is the matrix $(a_{i,j})_{1\leq i,j\leq n}$ with integer coefficients such that $a_{i,j} = k > 0$ if and only if there is an edge from vertex i to vertex j and this edge is the k-th outgoing vertex of i. Otherwise $a_{i,j} = 0$.*

An example DOAG and its transition matrix are pictured in Figure 2.6 on page 71, the meaning of the thick line and of the grey cells will be explained later. Let ϕ denote the function mapping a DOAG to its transition matrix. This function is clearly injective. In the rest of this section we will rely on this representation to work on DOAGs and provide asymptotic results. We begin with the characterisation of the image of ϕ.

First, observe that by definition of the traversal of the DOAG, the labelled transition matrix of a DOAG is strictly upper triangular. Moreover, the non-zero values of row i encode the *ordered* set of outgoing edges of vertex i. We state below a property of these rows. Note that in column j,

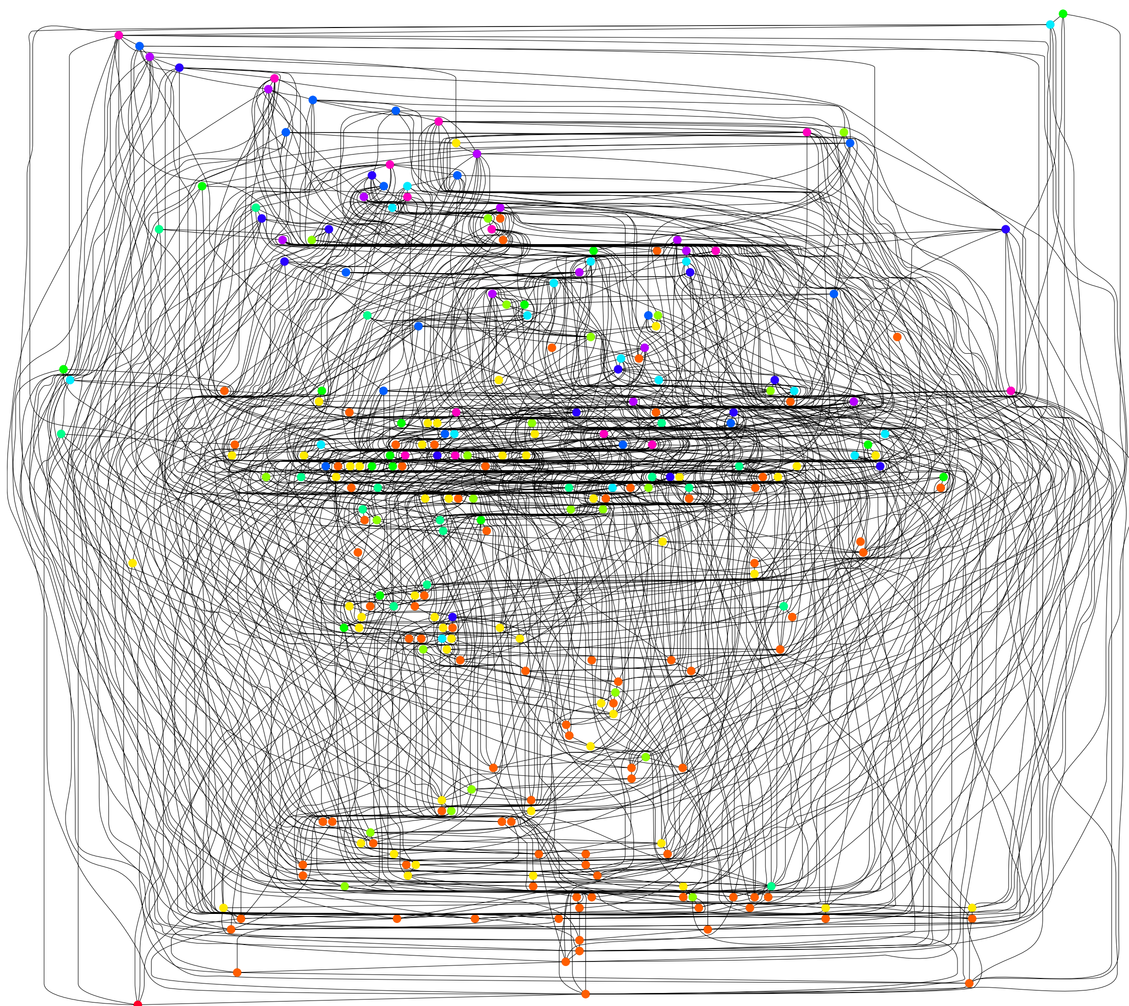
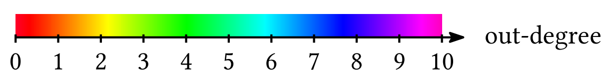


Figure 2.5: A random DOAG sampled uniformly at random among all DOAGs with $m = 1000$ edges and with maximum out-degree bounded by 10, that is such that all vertices have at most 10 outgoing edges. This DOAG contains 272 vertices. The colours of the vertices represent their out-degrees and have been picked according to the following colour map (lowest degree on the left and highest degree on the right).



the non-zero element with the *highest* index i, that is in the lowest position on the picture, has a spacial role: it corresponds to the last edge pointing to j when decomposing the DOAG. These elements are highlighted by a grey background in Figure 2.6 on the next page. As a consequence, when such cells occur on the same line i in the matrix, this means that when removing the i-th vertex in the decomposition, we uncover several new sources at once. By definition of the traversal, these sources are ordered according to the total order of the outgoing edges and thus,

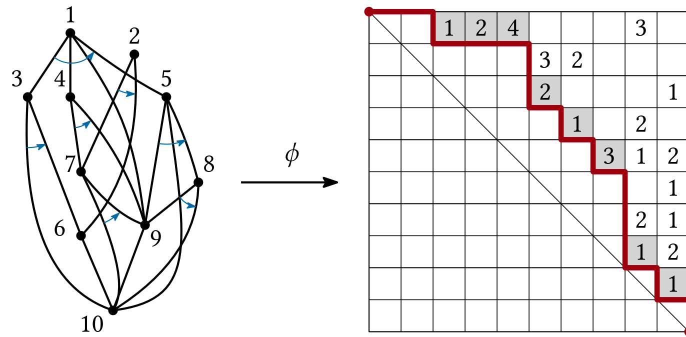


Figure 2.6: An example DOAG and its labelled transition matrix, the zeroes are not represented.

the values of these cells of the matrix must be ordered increasingly from left to right. For instance, observe that there are three grey cells in the first row of the matrix in Figure 2.6. Indeed, when removing the first source of the DOAG on the left, we uncover three new sources which are respectively in first, second and fourth position in the children of the removed source. Hence, the three grey cells in the first row of the matrix are 1, 2 and 4 which are indeed increasing.

Another property of the labelled transition matrix of a DOAG is that if there are several grey cells in the same row, they are necessarily consecutive. Indeed, these cells corresponds to edges pointing to newly uncovered sources upon removing one vertex of the DOAG during its traversal. By definition, these vertices get consecutive indices.

The outcome of the above discussion is summarised below and is actually the most difficult part of the characterisation the image of ϕ.

Proposition 4. *Let $A = (a_{i,j})_{1 \le i,j \le n}$ be the labelled transition matrix of some DOAG. For all $j \in [\![1; n]\!]$, let b_j denote the largest $i \le n$ such that $a_{i,j} > 0$ if such an index exists and 0 otherwise. We have that*

- *the sequence $j \mapsto b_j$ is weakling increasing;*
- *whenever $b_j = b_{j+1}$, we have that $a_{b_j,j} < a_{b_j,j+1}$.*

Proof. For proving the first point, one must observe that for all j, b_j is the step at which the j-th vertex becomes a source. As a consequence, since the sources are processed by the decomposition in the same order as they are discovered, the sequence $j \mapsto b_j$ is necessarily weakly increasing.

The second point is a consequence of the above discussion: two vertices which become sources at the same time get labels in the same order as their position as children of their parent. □

The thick line in Figure 2.6 is the path obtained by drawing horizontal lines at coordinates (b_j, j) for all j and connecting these lines using vertical lines. Proposition 4 states that the cells that are above the horizontal steps of this line must contain non-zero values and that consecutive such cells should contain increasing values.

Theorem 14. *Let $A = (a_{i,j})_{1 \le i,j \le n}$ be a strict upper triangular matrix of natural integers. Then A is the labelled transition matrix of a DOAG if and only if*

- *each row contains at least one non-zero number;*
- *each non-zero number is used at most once in a given row;*
- *the set of the non-zero numbers of a row form an interval whose lower bound is* 1*;*
- *Proposition 4 holds.*

Moreover, the number of sources of the DOAG whose image by ϕ is A is the number of leading zeros on the first row of A, and its number of edges is the number of non-zero values in the matrix.

The first three conditions of the theorem can be reformulated as: for each $i \in [\![1; n-1]\!]$, the sequence $(a_{i,j})_{i<j\leq n}$ is a *variation* of size $(n-i)$. Variations are defined and studied, from a combinatorial and a random-sampling perspectives, in Section 3.3.

Proof of Theorem 14. By definition, the labelled transition matrix of a DOAG satisfies the three first conditions of the theorem and the fourth has been proven in Proposition 4.

There remains to prove that the inverse transformation is possible. Given a matrix satisfying the four conditions of the theorem, we build a directed ordered graph as follows:

- consider the set $V = [\![1; n]\!]$ as a set of vertices;
- for each $v \in V$, put an edge from v to u if and only if $a_{v,u} > 0$ in the matrix and order the outgoing edges of v so that $(v, u) \prec_v (v, u')$ if and only if $a_{v,u} < a_{v,u'}$;
- Finally, order the sources of the graph so that $v \prec_\emptyset v'$ if and only if $v < v'$ as integers.

The directed graph described here is acyclic since the matrix is upper-triangular. Moreover, it is easy to check that its transition matrix is exactly the initial matrix, which concludes the proof. □

This characterisation of the labelled transition matrices of DOAGs gives a more *global* point of view on them compared to the decomposition given earlier, which was more *local*. In particular this helps establishing simple, though precise, lower and upper bounds on their number. Let $D_n = \sum_m D_{n,m,1}$ denote the number of DOAGs with one source and any number of edges. A trivial upper bound on D_n is given by the number of upper-triangular matrices satisfying all the conditions of Theorem 14 except for the last one. Said differently, the number of DOAGs of size n is upper-bounded by the number of $(n-1)$-uples of variations of sizes $1, 2, \ldots, (n-1)$. We have the number of variations of size k is given by $v_k = k! \sum_{p=0}^{k-1} \frac{1}{p!} \leq e \cdot k!$ (see Section 3.3 for instance), hence

$$D_n \leq \prod_{i=1}^{n-1} v_{n-i} \leq {¡}n-1! \cdot e^{n-1} \tag{2.2}$$

where ${¡}k! = \prod_{i=0}^{k} i!$ denotes the *super factorial* of k. The term "super factorial" seems to have been coined by Sloane and Plouffe in [SP95, page 228].

Obtaining a lower bound on D_n requires to find a subset of the possible labelled transition matrices described in Theorem 14 that is both easy to count and large enough to capture a large proportion of the DOAGs. One possible such set is that of the labelled transition matrices which have non-zero values on the super-diagonal $(a_{i,i+1})_{1\leq i<n}$. In such matrices, the constraints explained in Proposition 4 are trivially satisfied and this leaves a lot of free space on the right of

that diagonal to encode a large number of possible DOAGs. In such a matrix, the i-th row contains $p \leq n - i$ non-zero values, which can be in any position provided that one of them occupies the position $(i + 1)$. Hence, the number of such possible rows is

$$\begin{aligned}\sum_{p=1}^{n-i} \binom{n-i-1}{p-1} p! &= (n-i-1)! \sum_{p=0}^{n-i-1} \frac{n-i-p}{p!} \\ &= v_{n-i} - v_{n-i-1} \\ &= e \cdot (n-i-1)(n-i-1)! + o(1).\end{aligned}$$

As a consequence we have

$$D_n \geq \prod_{i=1}^{n-1} (v_i - v_{i-1}) \geq \frac{A}{n} \cdot ¡n-1! \cdot e^{n-1} \quad \text{for some } A > 0. \tag{2.3}$$

Although they are not precise enough to obtain an asymptotic equivalent for the sequence D_n, these two bounds already give us a good estimate of the behaviour of D_n. First of all, they let appear a "dominant" term of the form $¡n-1!$, which is uncommon in combinatorial enumeration. And second, it tells us we only make a relative error of the order of $O(n)$ when approximating D_n by $¡n-1! \cdot e^{n-1}$. Numerical experiments allow to conjecture that the behaviour of D_n is strictly between these two bounds. Before presenting these experimentations we give a recurrence formula that is more efficient to implement that the general formula on $D_{n,m,k}$ presented in Section 2.2.1 since we do not account for the number of edges here.

Let $D_{n,k}$ denote the number of DOAGs with n vertices (including k sources and one sink) and any number of edges. Using the same decomposition as above and applying the same combinatorial arguments we get

$$\begin{aligned}D_{n,k} &= \sum_{i+s>0} D_{n-1,k-1+s} \binom{s+i}{s} \binom{n-k-s}{i} i! \\ &= \sum_{s \geq 0} D_{n-1,k-1+s} \cdot \gamma(n-k-s, s)\end{aligned}$$

where

$$\gamma(a,b) = \mathbb{1}_{\{b \neq 0\}} + \sum_{i>0} \binom{b+i}{b} \binom{a}{i} i!$$

The above sum gives an explicit way to compute γ, but there is a computationally more efficient way to do so using recursion and memoisation:

$$\begin{aligned}\gamma(a,b) &= 0 && \text{when } a < 0 \text{ or } b < 0 \\ \gamma(0,b) &= \mathbb{1}_{\{b>0\}} \\ \gamma(a,b) &= \gamma(a, b-1) + a \cdot \gamma(a-1, b) + \mathbb{1}_{\{b=1\}} + a \cdot \mathbb{1}_{\{b=0\}} && \text{otherwise.}\end{aligned} \tag{2.4}$$

Using this recurrence formula with memoisation, the numbers $D_{n,k}$ for all $n, k \leq N$ can be computed in $O(N^3)$ arithmetic operations on big integers. Note that the D_n sequence defined above corresponds to $D_{n,1}$. Using the numbers computed by this algorithm, we plotted the

first 200 values of the sequence $n \mapsto \dfrac{D_n \sqrt{n}}{¡n-1!e^{n-1}}$ which seems to converge quickly to a constant. The plot is given in Figure 2.7. We do not have an explanation for the $\sqrt{n}$ term besides the fact that a plot in log-log scale of the sequence $\frac{D_n}{¡n-1!e^{n-1}}$ suggested to normalise by this value.

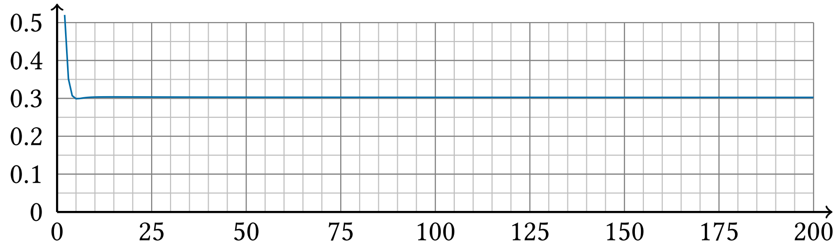


Figure 2.7: The first values of the sequence $n \mapsto \dfrac{D_n \sqrt{n}}{¡n-1!e^{n-1}}$.

We can thus state the following conjecture on the asymptotic behaviour of D_n.

Conjecture 1. *Asymptotically we have*

$$D_n \sim \frac{C}{\sqrt{n}} \cdot ¡n-1! \cdot e^{n-1}$$

for some constant $C \approx 0.30256$.

2.2.5 Uniform sampling of DOAGs by vertices only

The knowledge from the previous section on the asymptotic number of DOAGs with n vertices can be interpreted combinatorially to devise an efficient uniform random sampler of DOAGs based on rejection. The main idea of this section is that the upper bound we described on D_n corresponds to counting objects that are computationally cheap to sample, namely variations. Hence, a possible approach to sample uniform DOAGs is to sample uniform matrices made of $(n-1)$ variations of sizes $1, 2, \dots, (n-1)$ and reject them when they do not correspond to valid DOAGs. The probability of rejection of such a sampler is $1 - \dfrac{D_n}{¡n-1! \cdot e^{n-1}}$ and thus, the expected number of rejections is provably linear by (2.3) and is only of the order of $\sqrt{n}$ if Conjecture 1 is true.

Definition 14 (Variation matrix)**.** *We call a* variation matrix *of size n a strict upper triangular matrix $(a_{i,j})_{1\le i,j\le n}$ of non-negative integers such that for all $1 \le i < n$, the sequence $(a_{i,j})_{i<j\le n}$ is a non-empty variation (of size $n-i$).*

As already presented above, all the labelled transition matrices of DOAGs are variation matrices, and the number of such matrices of size n is given by:

$$A_n = \prod_{k=1}^{n-1} v_k = \Theta(¡n-1! \cdot e^{n-1}) = O(nD_n) \tag{2.5}$$

In the rest of this section, we describe a uniform random sampler of variation matrices. The first key step towards this goal, is to describe a uniform random sampler of variations. This is done separately in Section 3.3. Given a sampler of variations, we can then easily obtain a sampler of variation matrices by calling the sampler with the arguments $1, 2, 3, \dots, n-1$ and arranging the results in a matrix. This is presented in Algorithm 16.

Algorithm 16 Uniform random sampler of variation matrices

Input: An integer $n > 0$
Output: A uniform variation matrix of size n

```
function UnifVarMat(n)
    A = (a_{i,j})_{1≤i,j≤n} ⟵ a zero-filled n × n matrix
    for i from 1 to n do
        T ⟵ UnifVariationR(n − i)
        copy T to (a_{i,j})_{i<j≤n}
    return A
```

Finally, deciding whether an arrangement matrix is a valid labelled transition matrix can be implemented in quadratic time in n. We give such a decision procedure in Algorithm 17. In fact, in order to be more efficient, the array b computed in Algorithm 17 should be computed directly inside Algorithm 16 to avoid the extra traversal of the array. We keep it separated here for the sake of clarity.

Algorithm 17 Procedure deciding whether an arrangement matrix is a valid labelled transition matrix

Input: An arrangement matrix $A = (a_{i,j})_{1\le i,j\le n}$
Output: **true** if and only if A is a labelled transition matrix

```
function IsLabTransitionMatrix(A)
    b ⟵ zero-filled array of length n
    for i from 1 to n do
        for j from i + 1 to n do
            if a_{i,j} > 0 then
                b[j] ⟵ i
    for j from 1 to n − 1 do
        if b[j] > b[j + 1] then return false
        if (b[j] = b[j + 1] > 0) ∧ (a_{b[j],i} > a_{b[j],i+1}) then return false
    return true
```

Combining Algorithm 16 with a rejection procedure based on Algorithm 17, we finally get a uniform random sampler of labelled transition matrices, as presented in Algorithm 18. Theses matrices can then be transformed to DOAGs by applying ϕ^{-1}. To give a rough idea of the performance of this algorithm, our implementation allows to sample DOAGs of size $n = 2000$ in a few seconds on a standard laptop. Note that a DOAG of size n is actually a matrix of n^2 elements. Hence, for $n = 2000$ the "real" size of the sampled object is of the order of the million and the

complexity in terms of memory access of such a sampler is lower-bounded by $\Omega(n^2)$. Also note that we have $\log_2(D_n) \sim n^2 \log_2(n)$. As a consequence, the complexity in terms of random bits consumption of an uniform random sampler of DOAGs is in $\Omega(n^2 \ln(n))$. An open question regarding our sampler is: can we get closer to these bounds, for instance by checking the validity of the matrix on the fly and using early rejection?

Algorithm 18 Uniform random sampler of DOAGs with n vertices

Input: A positive integer n
Output: A uniform random DOAG with n vertices

```
function UnifDOAGn(n)
    A ← UnifArgMat(n)
    while not IsLabTransitionMatrix(A) do
        A ← UnifArgMat(n)
    return φ^{-1}(A)
```

2.3 Vertex-by-vertex decomposition of labelled DAGs

In this section we demonstrate the applicability of our method by establishing a new counting formula for the classical model of vertex-labelled DAGs with k sources and one sink. This corresponds to a sequence obtained by Gessel in [Ges96] using generating functions. The difference here is that our formula does not make use of the inclusion-exclusion principle and is thus amenable to effective random sampling. To our knowledge, this is the first such formula for labelled DAGs.

Another possible application of our method would be to count DAGs labelled on their *edges* rather than their *sources*. Such a labelling induces a natural ordering to the outgoing edges of each vertex, which allows to apply the same decomposition as in the previous sections. We do not detail this here for brevity and focus on vertex-labelled DAGs, which we believe are of greater interest.

2.3.1 Recursive decomposition

In a first attempt to decompose regular vertex-labelled DAGs, one might be tempted to devise a decomposition similar to DOAGs by removing the *smallest* source at each step. However, in this case this makes the recurrence difficult to express. Instead we count vertex-labelled DAGs with a *distinguished* source (this operation is called pointing), which makes the decomposition much simpler as we do not have to maintain an ordering. As for DOAGs, we only consider DAGs with *one* sink to ensure that they remain weakly connected.

Let $V_{n,m,k}$ denote the number of vertex-labelled DAGs with n vertices (including k sources and the unique sink) and m edges. The number of such DAGs with a distinguished (or pointed) source is given by $k \cdot V_{n,m,k}$ since any of the k sources may be distinguished. Let D denote one such DAG and let v denote its distinguished source. Removing the distinguished source in D yields

a regular vertex-labelled DAG D' with $n - 1$ vertices. Moreover, the three pieces of information that are necessary to reconstruct the source are the following:

1. the label of the source v which has been removed;
2. the set S of sources of D' which have been uncovered by removing v;
3. the set I of internal (non-sources) vertices of D' that were pointed to by an outgoing edge of v.

The reconstruction is then straightforward as one simply has to create a new (distinguished) source v with edges to S and I. This leads to the following recursive formula where $p = i + s$ denote the out-degree of v.

$$\begin{aligned} V_{1,m,k} &= \mathbb{1}_{\{m=0 \wedge k=1\}} & \\ V_{1,m,k} &= 0 & \text{when } k \leq 0 \\ k \cdot V_{n,m,k} &= n \sum_{p=1}^{\min(n,m)} \sum_{q=0}^{p} V_{n-1,m-p,k-1+q} \binom{n-q-k}{p-q} \binom{k-1+q}{q} & \text{otherwise.} \end{aligned} \tag{2.6}$$

Computing the first terms of this sequence with $k = 1$, we get back that values from A165950 related to the papers [Ges95; Ges96] where the sequence is obtained via a generating function enumeration related to the Tutte polynomial. The complexity of the straightforward dynamic programming algorithm implementing this formula to compute the first terms of $V_{n,m,k}$ is similar to that of DOAGs from the previous section.

Theorem 15. *Let $N, M > 0$ be two integers. Computing $V_{n,m,k}$ for all $n \leq N$, $m \leq M$ and all possible k can be done with $O(N^4 M)$ multiplications of integers of size at most $O(N^2)$.*

Proof. The only difference with DOAGs is in the size of the integers at play. As a rough upper-bound on the numbers $V_{n,m,k}$ is obtained by considering the total number of labelled DAGs with n vertices, that is $O\left(n! 2^{\binom{n}{2}} \rho^{-n}\right)$ for some constant $\rho > 0$. Taking the logarithm of this number yields that $O(N^2)$ bits are enough for storing the numbers we manipulate here. Moreover, better estimations of the number of DAGs are given in [BRRW86], which proves, in particular, that when $m \sim \binom{n}{2} \frac{1}{2}$ the logarithm of the number of DAGs with n vertices and m edges is lower-bounded by $\Omega(n^2)$. □

2.3.2 Random generation

A random sampling algorithm similar to Algorithm 14 can be obtained from formula (2.6). The major difference with the previous section is that one has to deal with the marking of the sources here and thus the division by k at the third line of (2.6). In fact it can be handled, at every recursive call, by first generating a DAG with a distinguished source (counted by $k \cdot V_{n,m,k}$) and then forgetting which source was distinguished. Since the recursive formula for $k \cdot V_{n,m,k}$ has no division, the uniform sampler of marked DAGs is obtained in a similar way as in the previous section. Moreover, forgetting which source was marked does not introduce bias in the distribution since all sources have the same probability to be marked. As side node: a similar technique (first pointing and then forgetting the pointed node) will be used in Chapter 3 in the

Algorithm 19 Uniform random sampler of vertex-labelled DAGs.

Input: Three integers (n, m, k) such that $V_{n,m,k} > 0$
Output: A uniform random vertex-labelled DAG with n vertices (including k sources and one sink), and m edges

```
function UnifLDAG(n, m, k)
    if n ≤ 1 then generate the (unique) DAG with 1 vertices
    else
        pick (p, s) with probability V_{n−1,m−p,k−1+s} binom(n−k−s, p−s) binom(k−1+s, s) / V_{n,m,k}
        D′ ⟵ UnifLDAG(n − 1, m − p, k − 1 + q)
        I ⟵ a uniform subset of size (p − s) of the inner vertices of D′
        Q ⟵ a uniform subset of size s the sources of D′
        pick a uniform ℓ ∈ ⟦1; n⟧
        relabel D′ by adding one to all of its labels ℓ′ such that ℓ′ ≥ ℓ
        add a new source to D′ labelled ℓ and with S ∪ I as its outgoing edges
        return D′
```

context of Boltzmann sampling. A uniform random sampler of vertex-labelled DAGs with n vertices (including k sources and one sink) and m edges is described in Algorithm 19.

This algorithm being very similar to Algorithm 14, it has the same complexity in terms of arithmetic operations on big integers, that is $O\big(\sum_v d_v^2\big)$ where v ranges over the vertices of the resulting graph and d_v denotes the out-degree of v.

2.4 Multi-graphs

We study here a variant of DOAGs where multi-edges are allowed, that is where there might be several edges with the same source and destination.

Definition 15 (DOAMG)**.** *A directed ordered multi-graph is a triple $(V, E, (\prec_v)_{v \in V \cup \{\varnothing\}})$ where:*

- *V is a finite set of vertices;*
- *$E : V \times V \longrightarrow \mathbb{N}$ is a finite multi-set of edges;*
- *for all $v \in V$, $\prec_v$ is a total order over the* multi-set *of* outgoing *edges of v, that is a finite sequence $(u_1, u_2, \dots, u_{d_v})$ such that for all $u \in V$, the number of occurrences of u in the sequence $\prec_v$ is the value of $E((v, u))$;*
- *and $\prec_\varnothing$ is a total order over the set of* sources *of the multi-graph, that is the vertices without any incoming edge.*

Two such multi-graphs are considered to be equal *if there exists a bijection between their respective sets of vertices that preserves the edges and the orders $\prec_v$. Finally, a DOAMG is a directed ordered multi-graph $(V, E, (\prec_v)_{v \in V \cup \{\varnothing\}})$ with only one sink and such that (V, E), as a directed multi-graph, is acyclic.*

DOAMGs are a generalisation of DOAGs in the sense that all DOAGs are DOAMGs. This kind of objects is suitable for representing partially compacted tree-like structures, typically appearing when using hash-consing [Ers58; Got74]. Contrarily to DOAGs, these objects may

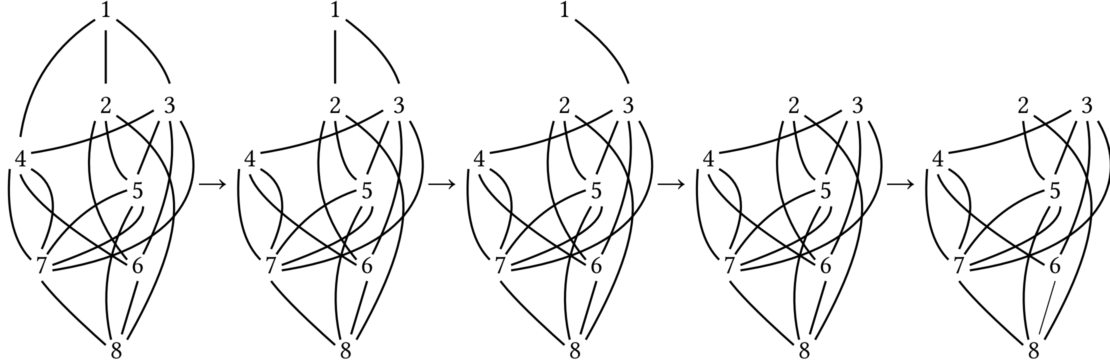


Figure 2.8: Example DOAMG and the first steps of its decomposition. The tiny arrows indicating the order of the outgoing edges are omitted for clarity, all outgoing edges are implicitly ordered from left to right.

have multiple edges between the same two vertices, which may naturally occur in this context. An example DOAMG is pictured in Figure 2.8.

2.4.1 Edge-by-edge decomposition scheme

We describe a similar decomposition as for DOAGs, starting from the source and removing one source after the other in an order dictated by the ordering of the outgoing edges of each vertex. However, in order to obtain a recurrence formula, we adopt here a "finer grained" decomposition by removing each source "edge-by-edge" rather than all at once. More formally, we describe what is left of such a graph if one removes the smallest edge of its smallest source. The edge-by-edge decomposition of these objects is the bijection defined as follows. A DOAMG D with at least two edges is mapped to a triple (D', v, b) such that:

- D' is a DOAMG obtained from D be removing the smallest outgoing edge of the smallest source u of D', if u had out-degree 1 in D, it is removed from D';
- v is the vertex of D' to which the removed edge was pointing;
- b is a Boolean equal to true if and only if u had degree 1 in D (and thus has been removed).

It is clear that this mapping is injective and one can check that any triple of the form (D', v, b) such that v is either an internal vertex of D' or is the largest source of D' is the image of a DOAMG by this mapping. It follows that the number $D'_{n,m,k}$ of DOAMGs with n vertices (including k ordered sources) and m edges satisfies the following recurrence relation. The first two terms of the second equality in (2.7) correspond to the case where v is an internal vertex of D' (there are $(n-k)$ such vertices, hence the $(n-k)$ factor) and the two other terms correspond to the cases where v is the largest source of D'. Depending on whether the smallest source of D had degree 1 or not, a source is removed from D in D'.

$$\begin{aligned} D'_{n,1,k} &= \mathbb{1}_{\{n=2 \wedge k=1\}} \\ D'_{n,m,k} &= (n-k)(D'_{n,m-1,k} + \mathbb{1}_{\{k>1\}} D'_{n-1,m-1,k-1}) + D'_{n-1,m-1,k} + D'_{n,m-1,k+1}. \end{aligned} \tag{2.7}$$

This formula can be used to compute all the coefficients $D'_{n,m,k}$ up to some bounds N and M in polynomial time using a dynamic programming approach. The procedure is straightforward and is not detailed here. The complexity of such a procedure is given in Theorem 16.

Theorem 16. *Let $N, M > 0$ be to integers. Computing $D'_{n,m,k}$ for all $n \leq N$, $m \leq M$ and for all possible k can be achieved in $O(N^2M)$ arithmetic operations. Moreover, the bit-complexity of these operations is at most $O(M \cdot \mathcal{M}(\log_2(N)))$ where $\mathcal{M}(x)$ is the bit-complexity of the multiplication*[2] *of two integers of bit-size x.*

Proof. There are $O(N^2M)$ coefficients $D'_{n,m,k}$ such that $n, k \leq N$ and $m \leq M$ and computing each one of them requires a constant number of arithmetic operations. So the first part of the theorem is clear.

Estimating the cost of the arithmetic operations requires to establish an upper bound on the numbers at play. By induction we have that $D'_{n,m,k} \leq (4N)^m$ and thus $\log_2(D'_{n,m,k}) \leq 2M + M\log_2(N)$. Then, we can observe that the only arithmetic operations in (2.7) are additions and multiplications of a small integer by a large integer. The additions have a linear cost in the size of their operands, that is $O(M\log_2(N))$. For the case of the multiplication, we have that $\log_2(n-k) \leq \log_2(N)$ and $\log_2(D_{n,m,k}) = O(M\log_2(N))$, hence, by Lemma 1 on page 6, they have a cost of at most $O(M \cdot \mathcal{M}(\log_2(N)))$. □

2.4.2 Random sampling

The above decomposition of DOAMGs (2.7) naturally translates into a recursive random sampler. Recall that the formula is based on a case analysis of the different scenarios when one removes the smallest edge of the smallest source. As a consequence, the algorithm follows a similar pattern, that is add *one* edge to a DOAMG sampled recursively of one of the four types described in the decomposition (and with the appropriate probability). The four types correspond to choosing whether the edge should point to an internal vertex or a source and whether it should be added to the smallest source or to a new source. This is described more precisely in Algorithm 20.

The complexity analysis of this algorithm is rather straightforward as it makes a linear number of recursive calls, each triggering the generation of one integer and a constant number of arithmetic operations.

Theorem 17. *Algorithm 20 called with the parameters n, m and k, makes $O(m)$ arithmetic operations of bit-complexity $O(m\log_2(n))$ and $O(m)$ calls the* Unif function.

Proof. The algorithm makes m recursive calls and the cost of the arithmetic operations has already been analysed in Theorem 16. □

2.4.3 Bijection with decorated north-east paths

Here we establish a bijection, inspired by (2.7), between DOAMGs and a class of pairs of walks in the quarter plane. Although we have not established any asymptotic results yet on this class, we

[2] See the generalities Chapter page 6

Algorithm 20 Uniform recursive random sampler of DOAMGs

Input: Three integers n, m and k such that $D'_{n,m,k} > 0$
Output: A uniform DOAG with n vertices (including k sources) and m edges

```
function UnifDOAMG(n, m, k)
    if m = 1 then return the unique DOAMG with 1 edge
    x ⟵ Unif([[0; D'_{n,m,k} − 1]])
    if x < (n − k)D'_{n,m−1,k} then
        D' ⟵ UnifDOAMG(n, m − 1, k)
        v ⟵ a uniform internal node of D'
        Add an edge from the smallest source of D' to v, in first position
    else if x < (n − k)(D'_{n,m−1,k} + D'_{n−1,m−1,k−1}) then
        D' ⟵ UnifDOAMG(n − 1, m − 1, k − 1)
        v ⟵ a uniform internal node of D'
        Add a new source to D' with a unique outgoing edge to v
    else if x < (n − k)(D'_{n,m−1,k} + D'_{n−1,m−1,k−1}) + D'_{n−1,m−1,k} then
        D' ⟵ UnifDOAMG(n − 1, m − 1, k)
        v ⟵ the largest source of D'
        Add a new source to D' with a unique outgoing edge to v
    else
        D' ⟵ UnifDOAMG(n, m − 1, k + 1)
        v ⟵ the largest source of D'
        Add an edge from the smallest source of D' to v, in first position
    return D'
```

believe this combinatorial interpretation can the starting point to such a study, and potentially write an improved sampler.

We first describe a mapping ϕ from the class of DOAMGs to the words on the infinite alphabet $\Sigma = \{\texttt{(},\texttt{)}\} \times \mathbb{N}$. For a DOAMG D with at least 2 edges, let $\mathrm{dec}(D) = (D', b, v)$ denote the results of its decomposition as described in Section 2.4.1. For any words w and w' on Σ, let ww' denote the concatenation of the two words. We define ϕ as follows.

$$\phi(D) = \begin{cases} \texttt{()} & \text{if } D \text{ has only one edge} \\ \phi(D')\phi_1(b)\phi_2(D', v) & \text{otherwise where } \mathrm{dec}(D) = (D', b, v) \end{cases} \tag{2.8}$$

where

- $\phi_1(\mathrm{true}) = \texttt{(}$;
- $\phi_1(\mathrm{false})$ is the empty word;
- if v is an interval vertex of D', $\phi_2(D', v)$ is the position of v, starting from 0, among the *internal* vertices of D', taken in the order induced by the decomposition;
- if v is the largest source of D', $\phi_2(D', v) = \texttt{)}$.

Note that there is some structure in the image of ϕ. For instance an instance of (is always followed by another letter which is different from (. Another property of these words is they are *almost* well-parenthesised in the sense that in every prefix, the number of opening parentheses

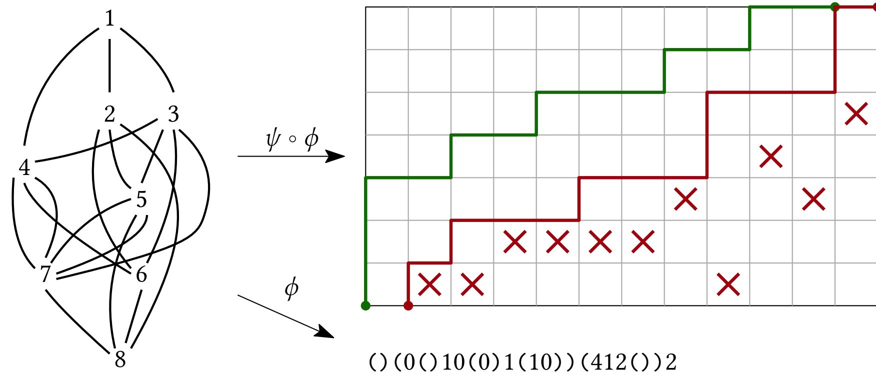


Figure 2.9: Example DOAMG, and its images by ϕ and $\psi \circ \phi$.

greater or equal to the number of closing parentheses. In addition, every integer appearing in a word must be preceded by more occurrences of) than its value. Actually, these properties characterise the image of ϕ, this is stated in Theorem 18.

Definition 16. *Let $\mathcal{W}$ denote the set of words w on the infinite alphabet $\{(,)\} \cup \mathbb{N}$ satisfying the following constraints:*

1. *w starts by* (;
2. if $w = w'w''$ then w' contains at least as many occurrences of (as occurrences of);
3. if $w = w'(w''$, then w'' is non-empty and does not start by (;
4. if $w = w'iw''$ for some $i \in \mathbb{N}$, then w' contains at least $(i + 1)$ occurrences of (.

Theorem 18. *The function ϕ defined in (2.8) is a bijection from the set of all DOAMG to the language $\mathcal{W}$ described in Definition 16. Moreover, if $\phi(D) = w$ and if n, m and k denote the number of vertices, edges and sources of D, then:*

- *w contains $n - 1$ opening parentheses* (;
- w contains $n - k$ closing parentheses);
- w has length $m + n - 1$.

Proof. The second property on the number of occurrences of (,) and the length of $\phi(D)$ is the result of a straightforward induction which is omitted here.

In order to prove that ϕ is a bijection, we define a function ϕ' from $\mathcal{W}$ to the set of DOAMGs and prove that this is the inverse of ϕ. First observe that because of point 3 of Definition 16, the last letter of a word of $\mathcal{W}$ is either an integer or (. We define the function ϕ' recursively by a case analysis on the last two letters of its argument and using the above observation. Let $w \in \mathcal{W}$ and let p and q denote respectively its number of (and).

- If $w = ()$, we define $\phi'(w)$ as the unique DOAMG with one edge.
- If w is of the form $w = w'(i$, then $w' \in \mathcal{W}$ since it cannot end with (so that we can take its image $D' = \phi'(w')$ by ϕ' (recursively defined). We have by induction that D' has p vertices and $(p - q)$ sources. We define $\phi'(w)$ as the DOAMG obtained by adding a new vertex to D' with only one edge to the i-th internal vertex of D'. This is well defined since D' has $p - (p - q) = q$ internal vertices and by point 4 of Definition 16 we have $i < q$.

- If w is of the form $w = w'()$, then $w' \in \mathcal{W}$ since it cannot end with (so that we can take its image $D' = \phi'(w')$ by ϕ' (recursively defined). We have by induction that D' has p vertices and $(p-q+1)$ sources. We define $\phi'(w)$ as the DOAMG obtained by adding a new vertex to D' with only one edge to the largest source of D'.
- If w is of the form $w = w'i$ and w' does not end with (, then $w' \in \mathcal{W}$ and we can take its image $D' = \phi'(w')$ by ϕ' (recursively defined). We have by induction that D' has $(p+1)$ vertices and $(p-q+1)$ sources. We define $\phi'(w)$ as the DOAMG obtained by adding a new edge in D' from its smallest source to its i-th internal vertex. This is well defined since D' has $(p+1)-(p-q+1) = q$ internal vertices and by point 4 of Definition 16 we have $i < q$.
- Finally, if w is of the form $w = w')$ and w' does not end with (, then $w' \in \mathcal{W}$ and we can take its image $D' = \phi'(w')$ by ϕ' (recursively defined). We have by induction that D' has $(p+1)$ vertices $(p-q+2)$ sources. We define $\phi'(w)$ as the DOAMG obtained by adding a new edge in D' from its smallest source to its largest source. This does not introduce a loop since D' has $(p-q+2) \geq 2$ sources (we have that $p \geq q$ by point 2 of Definition 16).

We have thus defined a function ϕ' from $\mathcal{W}$ to the set of DOAMGs, which is clearly injective. Finally, one can conclude the proof, either using by considering the restrictions of ϕ and ϕ' to objects with fixed parameters (number of vertices, edges, (, etc.) and invoking the finiteness of the sets, or by observing that ϕ and ϕ' are the inverse of each other. □

The bijection ϕ allows to encode DOAMGs as words. The properties of these words, which we have described in Definition 16, are well-suited for a more visual representation using decorated paths in the quarter plane. More precisely, we map the words of the image of ϕ to pairs of walks (W_1, W_2) in the quarter plane such that

- W_1 starts from $(0,0)$ and is made only of "north" $(0,+1)$ and "east" $(+1,0)$ steps;
- W_2 starts from $(1,0)$, is also made of north and east steps, and in addition the east steps are "decorated" with an integer;
- W_1 and W_2 have the same length and do not cross.

The mapping ψ from words to walks is described below as procedure. The auxiliary function ξ is necessary to alternate between one step on the first walk W_1 and one step on the second walk W_2.

$$\psi(w) = \begin{cases} \text{do nothing} & \text{if } w = \epsilon \\ \text{add a north step to } W_1 \text{ and execute } \xi(w') & \text{if } w = (w' \\ \text{add an east step to } W_1 \text{ and execute } \xi(w) & \text{otherwise} \end{cases}$$

$$\xi(w) = \begin{cases} \text{add a north step to } W_2 \text{ and execute } \psi(w') & \text{if } w =)w' \\ \text{add an east step, decorated with } i \text{, to } W_2 \text{ and execute } \psi(w') & \text{if } w = iw' \end{cases}$$

An example application of ψ is represented in Figure 2.9 on the previous page. Note that the properties of the words of $\mathcal{W}$ translate naturally to properties of the walks. For instance, in the figure, the integer i decorating each horizontal step of W_2 is represented as a cross in the "cell" located below it at distance i. One can prove that this integer i is smaller than the ordinate of the step is corresponds to. Also observe that the length of both walks is m, the final height of W_1 is $(n-1)$, and the final height of W_2 is $(n-k)$ where m, n and k are respectively the number of edges, vertices and sources of the original DOAMG. Moreover, another consequence

of Theorem 18 is that the walk W_1 must remain above of W_2 and their intersection must not contain any edge (but may contain single points).

Chapter 3

Algorithmic considerations related to random generation

This last chapter is dedicated to algorithmic and practical considerations regarding random samplers and unranking algorithms. We focus in particular on two aspects: practical efficiency and usability. Some of the results presented in this chapter (in Section 3.2) have been published in the paper [GP21][1] and the ideas exposed in Section 3.4 have lead to the implementation of a fast and generic Boltzmann sampling library [6] for Python.

We start this chapter by providing some historical context on the topic of random generation in Section 3.1. Then in Section 3.2, we study a basic combinatorial object, which is omnipresent as a building block for more complex data structures, namely: combinations. In particular we focus on the problem of the lexicographical unranking of combinations for which we propose two contributions. First we give a new point of view on the problem, based on the factoradic numeral system, allowing to see this problem as a special case of permutation unranking. And second, we propose to refine the complexity model that is traditionally used to analyse such algorithms. In light of this new model, we propose an optimisation that can be applied to all the algorithms from the literature as well as ours and which gives a significant performance improvement. In Section 3.3, we turn to the uniform random generation of variations[2]. Our contribution is a *linear* random sampler based on the rejection method which does not require the pre-computation of counting sequences like in the recursive method. Note that the choice of combinations and variations is not innocuous as they both play a central role in DAG enumeration and random sampling in Chapter 2. In fact, the complexity of some of our DAG samplers is directly linked to the complexity results from the present chapter.

Finally, in Section 3.4, we turn to Boltzmann sampling, which is a general purpose framework to write random samplers of combinatorial structures based on their specifications. This type of sampler has been used, in particular, to generate the random NFJ programs used in the experimentations from Chapter 1. Traditionally, this formalism is presented from a high level point of view and one of the contributions of this chapter is to dive deeper into the practical aspects of its implementation. In particular, we present an optimisation which has a significant

[1] [GP21] "Lexicographic unranking of combinations revisited" has been published in the Algorithms journal.

[2] Variations are also sometimes called k-permutation or arrangements

impact in practice as well as a formalism to give the user of such a library full control on how the generated objects are constructed in memory.

3.1 A review of random generation techniques

We give here an overview of some random sampling techniques and principles to set up the algorithmic context in which the present work takes place. As a consequence, we only give a restricted overview of the random sampling landscape and do not intend to be exhaustive. In particular, we focus on *uniform* random generation, which is a central aspect of this thesis.

We start by giving a classical example of a so-called *ad hoc* algorithm, designed for a specific combinatorial class. Although they are, by definition, very specific, such ad hoc algorithms form a solid background showcasing a wealth of techniques and ideas which can be applied, not only to other specific algorithms, but also to more general frameworks. The algorithm presented here introduces one such idea: the *rejection* principle.

Then we present the so-called "recursive method" from [NW78] which gives general principles to obtain uniform samplers for *decomposable* combinatorial structures. The idea is to follow a recursive decomposition of the structures to be generated into objects of smaller sizes. This approach relies on counting information on the objects at play, typically available under the form of sequences or generating functions, to weight some decision during the generation. Finally, we introduce the more recent framework of Boltzmann sampling [DFLS04], which takes a rather different approach. Here, the *value* of the generating functions (at some well-chosen real number) are used rather than their coefficients. Furthermore, Boltzmann samplers are able to generate larger structures thanks to their better complexity in exchange for a slight loss of precision of the size of the generated objects. Indeed, they generate structures of approximate size, that is *close* to a target size (rather than structures of exact size).

3.1.1 Ad hoc algorithms and the rejection method

In order to illustrate the rejection principle, we present a simplified version of the uniform random sampler of combinations of k elements among n from [NW78] (these are called k-subsets in the book). The problem is the following: how to sample, uniformly at random, a subset of size k of the set $[\![0; n-1]\!]$ of size n. A possible solution is to pick elements from this uniformly at random until k distinct ones have been found. This is described in Algorithm 21.

Algorithm 21 Ad hoc uniform random sampler of combinations

Input: Two integers n and k such that $0 \leq k \leq n$
Output: A uniform subset S of size k of $[\![0; n-1]\!]$

1: **function** UNIFCOMBINATION(n)
2: $\quad S \leftarrow \emptyset$
3: $\quad$ **while** $|S| < k$ **do**
4: $\quad\quad i \leftarrow$ UNIF($[\![0; n-1]\!]$)
5: $\quad\quad$ **if** $i \notin S$ **then** add i to S
6: $\quad$ **return** S

Note that this is a simpler algorithm than the one given in [NW78] because we ignore the cost of checking whether $i \in S$ at line 5 and the cost of adding one element to this set. Here, for an illustrative purpose, we only focus on the number of times this check is performed.

The rejection principle is materialised here in the fact that the new elements that are added to S are not sampled from $[\![0; n-1]\!] \setminus S$. Instead we pick an element of $[\![0; n-1]\!]$ uniformly at random and, if this element is in S already, we *reject* it and pick another one. An important question with such algorithms is: how much time do we waste in rejections? In this case, this corresponds to the coupon collector problem [Fel50, p. 213] which states that the expected number of trials is given by

$$E_n = n\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{n-k+1}\right) = n(H_n - H_{n-k+1})$$

where H_n denotes the n-th harmonic number. By symmetry of the problem, we can assume that $k \leq \frac{n}{2}$ (otherwise, it is more efficient to call the algorithm with $(n-k)$ instead of k and to take the complement of the result). In this case, since $H_n = \ln(n) + O(1)$, we have that

$$E_n = n \ln\left(\frac{n}{n-k+1}\right) + O(n) = O(n).$$

A large class of algorithms make use of this technique. For instance, it is used in our random sampler of variations in Section 3.3 (later in this chapter) and in the uniform DOAG sampler built on top of it in 2.2.5. This is also an essential component of Boltzmann sampling, a generic framework developed later in the history of random sampling and which is described in Section 3.1.4 below.

3.1.2 Recursive random sampling

An obvious drawback of ad hoc algorithms is that for every new combinatorial class, one must develop a new algorithm. It is natural to try to extract general principles from existing ad hoc algorithms to develop generic random sampling methods. This is the approach proposed by Nijenhuis and Wilf in [NW78]. After describing a few ad hoc algorithms in the first part of the book, the authors describe a general scheme that is applicable to all the objects presented before, and more. The common property shared by these objects, and which is at the core of the method is that they are *decomposable*.

We illustrate this again on the example of combinations. Let $\mathcal{A}_{n,k}$ denote the set of the combinations of k elements among n. An element of $\mathcal{A}_{n,k}$ either contains $n-1$, in which case its $k-1$ other elements form a combination of $k-1$ elements among the $n-1$ elements of $[\![0; n-2]\!]$, or does not contain $n-1$, in which case it belongs to $\mathcal{A}_{n-1,k}$. This leads to the following identity which "decomposes" $\mathcal{A}_{n,k}$ in two smaller classes:

$$\mathcal{A}_{n,k} = \mathcal{A}_{n-1,k-1} \times \{n-1\} + \mathcal{A}_{n-1,k}. \tag{3.1}$$

Note that the cardinality of $\mathcal{A}_{n,k}$ is given by the binomial coefficient $\binom{n}{k}$. The above identity thus allows to recover the well-known Pascal formula $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$. A recursive random sampler can be obtained from this decomposition. The idea is to make a random choice between the two sides of the disjoint union in (3.1) and to weight this choice using their cardinality. Concretely this means choosing $\mathcal{A}_{n-1,k-1} \times \{n-1\}$ with probability $\binom{n-1}{k-1}/\binom{n}{k} = \frac{k}{n}$ and $\mathcal{A}_{n-1,k}$ with

probability $\frac{n-k}{n}$. This is implemented in Algorithm 22 below by the function BERN generating a Bernoulli variable.

Algorithm 22 Recursive uniform random sampler of combinations

Input: Two integers n and k such that $0 \leq k \leq n$
Output: A uniform subset S of size k of $[\![0; n-1]\!]$

1: **function** UNIFCOMBINATIONREC(n, k)
2: **if** $k = 0$ **then return** $\emptyset$
3: **else if** $n = k$ **then return** $[\![0; n-1]\!]$
4: **else if** BERN$\left(\frac{k}{n}\right)$ **then return** UNIFCOMBINATIONREC($n-1, k-1$) $\cup \{n-1\}$
5: **else return** UNIFCOMBINATIONREC($n-1, k$)

A large variety of combinatorial structures admit such a decomposition. For instance, a permutation σ of $[\![0; n-1]\!]$ can be identified to a permutation σ' of $[\![0; n-2]\!]$ and the position in σ' at which $(n-1)$ should be inserted to obtain σ. A different kind of decomposition is used for the NFJ programs from Chapter 1: they are described by a context-free grammar. An NFJ program is either an atomic action (this is the base case) or is made of two smaller programs combined using one of the three operators of the language. Binary trees, plane trees, series-parallel circuits, etc. admit a similar kind of decomposition.

Automation

Whereas the general principles of the recursive method were given in [NW78], it took a significant leap forward with [FZV94] which formally describes a *systematic* way to obtain a uniform random sampler for any class admitting an combinatorial *specification*. With [FZV94], the recursive method moves from a set of general principles to the rank of a proper framework.

A key feature of these specifications is that they translate into relations between generating function which can be used to compute, systematically, the sequences of integers needed for the random generation. This is an essential pre-processing step. In addition, and more importantly, [FZV94] describes a set of rules to map each construction of a specification to an algorithm combinator. We give the more important ones here.

Disjoint union The simplest rule of the recursive method is that of the disjoint union. If a finite set $\mathcal{C}$ is defined as a disjoint union $\mathcal{A} + \mathcal{B}$, then for drawing a uniform element of $\mathcal{C}$ one has to draw a uniform element of $\mathcal{A}$ with probability $\frac{|\mathcal{A}|}{|\mathcal{C}|}$ and a uniform element of $\mathcal{B}$ otherwise. In terms of combinatorial classes, if $\mathcal{C} = \mathcal{A} + \mathcal{B}$ is a combinatorial class and if a_n, b_n and c_n denote the number of elements of size n of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, then a uniform element of $\mathcal{C}$ of size n must be a uniform element of size n of $\mathcal{A}$ with probability $\frac{a_n}{c_n}$. This translates to Algorithm 23.

Cartesian product The case of the Cartesian product $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ of two combinatorial classes builds on the same principle. If $\mathcal{C}_n$ (resp. $\mathcal{A}_n$, $\mathcal{B}_n$) denotes the set of elements of size n of $\mathcal{C}$ (resp. $\mathcal{A}$, $\mathcal{B}$), then the idea is to express $\mathcal{C}_n$ as follows:

$$\mathcal{C}_n = \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \cdots + \mathcal{A}_{n-1} \times \mathcal{B}_1 + \mathcal{A}_n \times \mathcal{B}_0.$$

Algorithm 23 Recursive method for the disjoint union

Input: A non-negative integer n
Output: A uniform element of size n of $\mathcal{C} = \mathcal{A} + \mathcal{B}$

```
function UnifC(n)
    if Bern(a_n / c_n) then return UnifA(n)
    else return UnifB(n)
```

If follows that in order to sample a uniform element of $\mathcal{C}_n$, one may use the same method as explained above to select $k \in [\![0; n]\!]$ and then a uniform element of $\mathcal{A}_k$ and a uniform element of $\mathcal{B}_{n-k}$, as detailed in Algorithm 24.

Algorithm 24 Recursive method for the Cartesian product

Input: A non-negative integer n
Output: A uniform element of size n of $\mathcal{C} = \mathcal{A} \times \mathcal{B}$

```
1: function UnifC(n)
2:     k ← drawn from [[0; n]] with probability P[k] = a_k b_{n-k} / c_n
3:     return (UnifA(k), UnifB(n − k))
```

The performance of Algorithm 24 is closely tied to the generation of the random variable k as line 2. A natural algorithm would be to sample a uniform integer r in $[\![0; c_n - 1]\!]$ and to find the minimum $k \geq 0$ such that $\sum_{i=0}^{k} a_i b_{n-i} > U$. This corresponds to considering the terms $\mathcal{A}_k \mathcal{B}_{n-k}$ in *lexicographic* order. However it has been shown in [FZV94] that a better complexity can be obtained by using the so-called *boustrophedonic* order, which consists in summing those terms in the following order $a_0 b_n + b_0 a_n + a_1 b_{n-1} + a_{n-1} b_1 + \cdots$. The two methods are given in Algorithm 25.

Algorithm 25 Generation of the index k for case of the Cartesian product

Using the lexicographic order

```
U ← Unif([[0; c_n − 1]])
S ← 0    k ← 0
while S ≥ a_k b_{n-k} do
    S ← S − a_k b_{n-k}
    k ← k + 1
```

Using the boustrophedonic order

```
U ← Unif([[0; c_n − 1]])
S ← 0    k ← 0
while S ≥ a_k b_{n-k} + a_{n-k} b_k do
    S ← S − a_k b_{n-k} − a_{n-k} b_k
    k ← k + 1
if S ≥ a_k b_{n-k} then  k ← n − k
return n − k
```

At the level of *one* Cartesian product, the two methods for drawing k are in fact equivalent. In both cases, n additions or subtraction and $n+1$ multiplications are made. The difference between the two lies in the recursive calls in the worst case. Using the boustrophedonic order, the worst case corresponds to $k = \frac{n}{2}$ whereas it is $k = n$ using the lexicographic order. This has an important impact when the above rules are used recursively to sample a structure described by a recursive specification. The *worst* case of the boustrophedonic order is associated to recursive calls to the sampler with $\frac{n}{2}$ as argument which mimics a "divide-and-conquer" scheme. This balances the

effect of the "costly" generation of k when this happens. Using this approach, sampling a uniform element of given size for a specifiable class is quasi-linear, as stated in Theorem 19.

Theorem 19 (Boustrophedonic random generation ([FZV94, Theorem 4.1]))**.** *Any decomposable structure has a random generation routine that uses pre-computed tables of size $O(n)$ and achieves $O(n \log(n))$ worst-case time complexity.*

The pre-computation mentioned by the theorem corresponds to computing the number of elements of size k of the classes involved for all $k \leq n$. The proof of the time complexity is given in [FZV94, page 15]. For the purpose of giving a quick overview of the recursive method, we have only given its most basic rules but the above theorem applies for a large class of combinatorial specifications using a wide range of operators. The full list is available in [FZV94].

Example: NFJ programs Finally, in order to illustrate this introduction, we apply it to the uniform random generation of syntactic NFJ programs without loops from Chapter 1. For convenience, we recall the specification of this class $\mathcal{F}$ below. Recall that the three terms of the form $\mathcal{F}\times\mathcal{F}$ respectively correspond to programs of the form $(P \parallel Q)$ (parallel composition), $(P+Q)$ (non-deterministic choice), and $(P;Q)$ (sequential composition). The $\mathcal{Z}$ term accounts for the atomic actions of the language.

$$\mathcal{F} = \mathcal{Z} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F}$$

The generating function $f(z)$ of the class $\mathcal{F}$ satisfies $f(z) = z + 3f(z)^2$. This equation can be used to compute the coefficients f_n of $f(z)$ recursively or, as we saw in Chapter 1 in Theorem 1, to establish the explicit formula $f_n = \frac{3^n}{12n-6}\binom{2n}{n}$, which we will use here. Applying the rules described in this section, a uniform sampler of NFJ programs is given by Algorithm 26.

Algorithm 26 Recursive sampler of NFJ programs

Input: A positive integer n
Output: A uniform NFJ program of size n

```
function UnifNFJ(n)
    if n = 1 then return a fresh atom
    else
        U ← Unif([[0; f_n^2 − 1]]),   i ← Unif([[1; 3]]),   S ← 0,   k ← 0
        while S ≥ 2 f_k f_{n−k} do
            S ← S − 2 f_k f_{n−k}
            k ← k + 1
        if S ≥ f_k f_{n−k} then k ← n − k
        if i = 1 then return (UnifNFJ(k) ∥ UnifNFJ(n − k))
        else if i = 2 then return (UnifNFJ(k) + UnifNFJ(n − k))
        else if i = 3 then return (UnifNFJ(k); UnifNFJ(n − k))
```

3.1.3 Unranking (and ranking)

A problem closely related to random generation is the *unranking* problem. Given a total order over the elements (say of given size) of a combinatorial class and given a rank u, it consists in answering the question: "what is the u-th element of this class?". Unranking allows for a simple form of random generation: rather than drawing a uniform object directly, draw a uniform rank u in the interval of valid ranks and unrank the object of rank u. Moreover, unranking algorithms describe bijections between families of combinatorial objects and integer intervals. This bijection can be used to represent combinatorial objects as integers, which is a more compact and sometimes handy format for storing and manipulating them. The inverse operation, which maps combinatorial structures to their rank, is called *ranking*.

The authors of [NW78] studied this question and proposed some algorithms for a few structures. Later, Molinero developed in [Mol05, Part II] a framework, similar to that of [FZV94] for the random generation problem, to rank and unrank combinatorial structures based on a combinatorial specification. Many practical aspects and performance heuristics are considered in this work. We present here briefly the basic rules for unranking unlabelled structures and the interested reader may refer to [Mol05] for a more thorough introduction.

Disjoint union Let $\mathcal{C} = \mathcal{A} + \mathcal{B}$ be a combinatorial class defined as the disjoint union of two other classes. If a total order is given on the elements $\mathcal{A}$ (resp. $\mathcal{B}$), then a total order on the elements of size n of $\mathcal{C}$ is obtained by considering that the elements of $\mathcal{B}$ are greater than the elements of $\mathcal{A}$. Using this order, a simple unranking procedure of $\mathcal{C}$ is given in Algorithm 27.

Algorithm 27 Unranking algorithm for the disjoint union

Input: An size n and a rank $0 \leq u < c_n$ where $c_n = |\mathcal{C}_n|$
Output: The u-th elements of $\mathcal{C}$

```
function UNRANKC(n, u)
    if u < a_n then return UNRANKA(n, u)
    else return UNRANKB(n, u − a_n)
```

Cartesian product Let $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ be the Cartesian product of two combinatorial classes. We saw in Section 3.1.2 that the decomposition $\mathcal{C}_n = \bigcup_{k=0}^{n} \mathcal{A}_k \times \mathcal{B}_{n-k}$ could be used to treat the Cartesian product $\mathcal{A} \times \mathcal{B}$ using a similar rule to the disjoint union case. It turns out that the two algorithms we proposed for the generation of the index k correspond, in the unranking world, to two distinct total orders on $\mathcal{C}$. The lexicographic algorithm corresponds to the lexicographic ordering where whenever $k < \ell$ then the elements of $\mathcal{A}_k \times \mathcal{B}_{n-k}$ are smaller than the elements of $\mathcal{A}_\ell \times \mathcal{B}_{n-\ell}$. Likewise, the boustrophedonic ordering corresponds to ordering the sets $\mathcal{A}_k \times \mathcal{B}_{n-k}$ like $\mathcal{A}_0 \times \mathcal{B}_n < \mathcal{A}_n \times \mathcal{B}_0 < \mathcal{A}_1 \times \mathcal{B}_{n-1} < \mathcal{A}_{n-1} \times \mathcal{B}_1 < \mathcal{A}_2 \times \mathcal{B}_{n-2} < \cdots$. As for the random sampling problem, the boustrophedonic order achieves better performance and should be used unless the lexicographic ordering is a requirement of the application using the algorithm. The unranking procedure for the boustrophedonic order is given in Algorithm 28.

Algorithm 28 Unranking procedure for the Cartesian product in the boustrophedonic order

Input: A size n and a rank $0 \leq u < c_n$
Output: The u-th element of $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ for the boustrophedonic order

```
function UnrankC(n, u)
    k ← 0
    while u ≥ a_k b_{n-k} + a_{n-k} b_k do
        u ← u − a_k b_{n-k} + a_{n-k} b_k
        k ← k + 1
    if u ≥ a_k b_{n-k} then
        u ← u − a_k b_{n-k}
        k ← n − k
    return (UnrankA(k, ⌊u / b_{n-k}⌋), UnrankB(n − k, u mod b_{n-k}))
```

3.1.4 Boltzmann sampling

The recursive method presented above, as well as the framework from [Mol05] for unranking combinatorial structures, relies on counting information in the form of sequences or generating functions. Efficient methods were described to implement this pre-processing (see [PSS12] in particular). However, the computation and the storage of the large integers at play limits the size of the objects that can be generated to a few thousand nodes in general. This motivated a change of paradigm in [DFLS04] which brought two novel ideas. First, rather than considering the *coefficients* of some generating functions, one may *evaluate* these functions at well-chosen points on the real axis, which is computationally cheaper. And second, one may generate much larger objects by relaxing the size constraint and allowing objects of size *close to* n rather than *exactly* n, which is enough for some applications like random testing for instance.

We recall here the principles of Boltzmann sampling and the main results from [DFLS04].

The Boltzmann model

The main idea of Boltzmann sampling is to draw objects from a combinatorial class according to the probability distribution given below, called the Boltzmann model. This comes by opposition to the usual paradigm which consists in sampling a uniform element of given size.

Definition 17 (Boltzmann model, Boltzmann sampler)**.** *Let $\mathcal{A}$ denote a combinatorial class and let $A(z)$ denotes its* ordinary *generation function (OGF). Let $z > 0$ such that A converges at z. The Boltzmann model of parameter z over $\mathcal{A}$ is defined by:*

$$\mathbb{P}_{\mathcal{A}}[a \in \mathcal{A}] = \frac{z^{|a|}}{A(z)}$$

where $|a|$ denotes the size of a. A Boltzmann sampler *$\Gamma[\mathcal{A}](z)$ for $\mathcal{A}$ is an algorithm sampling elements of $\mathcal{A}$ according to this probability distribution.*

The probability of an element of $\mathcal{A}$ depends only on its size and on the value of the parameter z. Thus, for a given value of z, two elements of the same size have the same probability of

being drawn. Besides this property, the Boltzmann model has two main interests which give it a special place in the random generation landscape. First, Boltzmann samplers have compositional properties similar to that of generating functions, which make them especially relevant for combinatorial classes admitting a *specification*. Moreover, Boltzmann samplers can be used to efficiently sample objects of size close to a desired value.

Remark 4. *We limit ourselves to the unlabelled model here in order to keep the presentation simple but everything presented in this section can be adapted to the labelled case.*

Composition rules

Since the Boltzmann model is closely tied to the generating function of the class at play, it composes well with the classical operators of analytic combinatorics. For instance, let $\mathcal{A}$ and $\mathcal{B}$ be two combinatorial classes and let $c = (a, b)$ be an element of their Cartesian product $\mathcal{C} = \mathcal{A} \times \mathcal{B}$. Then the probability of c is given by

$$\mathbb{P}_{\mathcal{C}}[c] = \frac{z^{|c|}}{C(z)} = \frac{z^{|a|+|b|}}{A(z)B(z)} = \mathbb{P}_{\mathcal{A}}[a] \cdot \mathbb{P}_{\mathcal{B}}[b].$$

As a consequence, it is straightforward to obtain a Boltzmann sampler for $\mathcal{C}$ given a Boltzmann sampler for $\mathcal{A}$ and $\mathcal{B}$: simply make two *independent* calls to the $\mathcal{A}$ and $\mathcal{B}$ samplers and return the two results as a pair.

Similarly if $\mathcal{C} = \mathcal{A} + \mathcal{B}$ is a disjoint union, then the probability of $c \in \mathcal{C}$ is given by:

$$\mathbb{P}_{\mathcal{C}}[c] = \frac{z^{|c|}}{C(z)} = \begin{cases} \frac{A(z)}{A(z)+B(z)} \mathbb{P}_{\mathcal{A}}[c] & \text{if } c \in \mathcal{A} \\ \frac{B(z)}{A(z)+B(z)} \mathbb{P}_{\mathcal{B}}[c] & \text{if } c \in \mathcal{B}. \end{cases} \tag{3.2}$$

Hence, a Boltzmann sampler for $\mathcal{A} + \mathcal{B}$ is obtained by generating a Bernoulli variable of parameter $\frac{A(z)}{A(z)+B(z)}$ and calling a Boltzmann sampler for either $\mathcal{A}$ or $\mathcal{B}$ depending on the outcome.

Similar composition rules can be obtained for the other usual operators of analytic combinatorics. Some of them are recalled in Table 3.1 and more are given in [DFLS04], [FFP07] and [BFKV11].

Table 3.1: Classical unlabelled operators of the symbolic method and their Boltzmann samplers $\Gamma[\mathcal{C}]$

Class	Spec.	OGF. $C(z)$	Boltzmann sampler $\Gamma[\mathcal{C}](z)$
Neutral	$\mathcal{E}$	1	ϵ
Atom	$\mathcal{Z}$	z	z
Union	$\mathcal{A} + \mathcal{B}$	$A(z) + B(z)$	**if** $\text{Bern}\left(\frac{A(z)}{A(z)+B(z)}\right)$ **then** $\Gamma[\mathcal{A}](z)$ **else** $\Gamma[\mathcal{B}](z)$
Product	$\mathcal{A} \times \mathcal{B}$	$A(z) \cdot B(z)$	$(\Gamma[\mathcal{A}](z), \Gamma[\mathcal{B}](z))$
Sequence	$\textsc{Seq}(\mathcal{A})$	$(1 - A(z))^{-1}$	$k \leftarrow \text{Geom}(1 - A(z)); (\Gamma[\mathcal{A}](z))_{1 \le i \le k}$
Multi-set	$\textsc{MSet}(\mathcal{A})$	$\exp\left(\sum_{j>0} \frac{A(z^j)}{j}\right)$	cf. Algorithm 29

Algorithm 29 Boltzmann sampler for the multi-set construction $\textsc{MSet}(\mathcal{A})$

> **function** $\Gamma[\textsc{MSet}(\mathcal{A})](z)$
> $M \longleftarrow$ empty multi-set
> $k_0 \longleftarrow$ drawn according to $\mathbb{P}[k] = \exp\left(\frac{1}{k}A(z^k)\right) / \exp\left(\sum_{j>0}\frac{1}{j}A(z^j)\right)$
>   **for** j **from** 1 **to** $k_0 - 1$ **do**
>   **for** i **from** 1 **to** $\textsc{Pois}\left(\frac{A(x^j)}{j}\right)$ **do**
> $\gamma \longleftarrow \Gamma[\mathcal{A}](z^j)$
> $M \longleftarrow M \cup \{\gamma, \gamma, \ldots, \gamma\}$ (γ duplicated j times)
>   **for** i **from** 1 **to** $\textsc{Pois}_{\geq 1}\left(\frac{A(x^{k_0})}{k_0}\right)$ **do**
> $\gamma \longleftarrow \Gamma[\mathcal{A}](z^{k_0})$
> $M \longleftarrow M \cup \{\gamma, \gamma, \ldots, \gamma\}$ (γ duplicated k_0 times)
>   **return** M

These composition rules can be used to write, in a systematic way, a Boltzmann sampler for any combinatorial class described by a combinatorial specification. For instance, we recall here the specification of the class $\mathcal{F}$ of NFJ programs without loops given in (1.1) in Section 1.2.2.

$$\mathcal{F} = \mathcal{Z} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F}. \tag{3.3}$$

It is straightforward to obtain a Boltzmann sampler for $\mathcal{F}$ by applying the rules from Table 3.1 on the previous page, and with the use of recursion to deal with the occurrences of $\mathcal{F}$ on the right-hand-side of (3.3). This sampler is given in Algorithm 30. We will keep using the class $\mathcal{F}$ as a running example in this section.

Algorithm 30 Example Boltzmann sampler for the class $\mathcal{F}$ described in (3.3)

> **function** $\Gamma[\mathcal{F}](z)$
> $U \longleftarrow \textsc{Unif}([0; f(z)])$
>   **if** $U < z$ **then return** a fresh atom
>   **else if** $U < z + f(z)$ **then return** $(\Gamma[\mathcal{F}](z) \parallel \Gamma[\mathcal{F}](z))$
>   **else if** $U < z + 2f(z)$ **then return** $(\Gamma[\mathcal{F}](z) + \Gamma[\mathcal{F}](z))$
>   **else return** $(\Gamma[\mathcal{F}](z); \Gamma[\mathcal{F}](z))$
>
> where $f(z) = \frac{1-\sqrt{1-12z}}{6}$.

Note that the Boltzmann samplers defined here can sample objects of any size, they operate "freely" under the influence of the parameter z and, as such, are called *free* samplers. They have linear complexity, in the size of their output, in terms of arithmetic operations on real numbers. We explain below how they can be constrained to sample objects of size close to a target size n and how to choose the value of z to maximise the probability to get such an object.

Approximate-size sampling

The usual way to generate structures close to a target size is to combine a free Boltzmann sampler with a rejection procedure. More precisely, here we consider given a target size n and a

tolerance $\epsilon > 0$, and we define a random sampler of structures whose size belongs to the interval $I(n,\epsilon) = [(1-\epsilon)n; (1+\epsilon)n]$ as described in Algorithm 31. We call such an algorithm a *rejection* sampler and the distribution of its outcome is the Boltzmann model conditioned to only yield objects of size in $I(n,\epsilon)$.

Algorithm 31 Rejection sampler for the class $\mathcal{A}$

Input: A target size n, a tolerance ϵ and a parameter z
Output: An random element of $\mathcal{A}$ of size in $I(n,\epsilon)$ in the Boltzmann model

```
function Γ[A](z, n)
    γ ← Γ[A](z)
    while |γ| ∉ I(n, ε) do γ ← Γ[A](z)
    return γ
```

The efficiency of rejection samplers depends on two major factors. First, the value of the parameter z must be chosen carefully so that the probability that a free Boltzmann sampler yields an object of size close to n is large. To this end, a good heuristic is obtained by computing the expectation $\mathbb{E}_z[N]$ of the size N of the outcome of a free sampler and solving the equation $\mathbb{E}_z[N] = n$. If $A(z)$ denotes the generating function of the class at play, a straightforward computation shows that $\mathbb{E}_z[N] = z\frac{A'(z)}{A(z)}$. This quantity is a non-decreasing function and, in the (rather common) case where it tends to ∞ when z tends to the radius of convergence ρ of $A(z)$, the equation $A(z)n = zA'(z)$ admits a solution for all n. The problem of solving this equation and computing the value of $A(z)$ at this point is often referred to as the "tuning" of the sampler and can be delegated to an external procedure called a "tuner" or an "oracle". The problem of writing an efficient and numerically stable oracle is a research problem in itself and several solutions have been proposed, notably a fast approximation procedure based on the Newton iteration method [Piv08; PSS12] and a reformulation of the problem as a convex optimisation problem [BBD18]. In our Boltzmann sampling library, we rely on existing implementations of these two techniques in [11] and [3].

Unfortunately, the choice of the value of z is not the only factor in the performance of a rejection sampler $\Gamma[\mathcal{A}](z,n)$ and the shape of the distribution of the size in the Boltzmann model of $\mathcal{A}$ is crucial too. This is closely tied to the generating function $A(z)$ and its behaviour near its dominant singularity ρ. The authors of [DFLS04] classify the possible distributions of the size in three categories:

- *bumpy* distributions for which the rejection sampler succeeds in one trial with high probability, in that case the complexity of approximate size sampling is $O(n)$ without further modification;
- *flat* distributions for which the rejection sampler succeeds in a constant number of trials in average, in that case the complexity of approximate size sampling is $O(n)$ too, this is typically the case for rational specifications;
- and finally, *peaked* distributions for which the rejection sampler is not linear as such because most of the weight of the distribution is concentrated on small sizes, this is typically the case for tree-like structures such as binary trees.

Two techniques are proposed in [DFLS04] to address the issue of the *peaked* case. A first technique consists in modifying the grammar using the so-called "pointing" operator so that the new

grammar belongs to the *flat* category. This basically corresponds to the considering the class $\mathcal{A}^\bullet$ of objects of $\mathcal{A}$ where one of the atoms of the structure is distinguished. This does not introduce any bias in the generation but has the effect of differentiating the generating function and hence changing the type of its dominant singularity. An alternative technique consists in choosing the special value $z = \rho$ as the parameter of the distribution to maximise the probability to generating large structures. In this case, one must also implement an *early* rejection procedure, that is to say that during the generation of an object, a global counter keeps track of the number of already generated atoms so that the generation can be aborted as soon as this number becomes larger than $(1 + \epsilon)n$.

The different cases briefly exposed here are discussed in depth in [DFLS04] and the overall conclusion of the article is that linear complexity can be achieved in all cases provided that the value of z is chosen carefully and the type of the distribution of the sizes is taken into consideration. In addition, a detailed comparative study of the two proposed solutions for the *peaked* case is done in [BGR15].

This concludes this quick overview of the random generation landscape. From now on and until the end of this last chapter of the thesis, we present our contributions to the field of random sampling.

3.2 Faster lexicographic unranking of combinations

In this section we dive into the topic of lexicographic combinations unranking. A combination of k elements among n is a subset of size k of $[\![0; n-1]\!]$. We represent these sets as increasing (finite) sequences of k elements so that we can compare them using the lexicographic order. We consider the problem of unranking combinations lexicographically, that is to say: “given two integers $0 \leq k \leq n$ and a rank $0 \leq u < \binom{n}{k}$, what is the u-th combination of k elements among n according to the lexicographic order?” The results we present in this section have been published in [GP21].

Combination unranking is an old problem, we start by giving an overview of the historical approaches and recall their main results in Section 3.2.1. Then, in Section 3.2.2, we present a new point of view on the problem based on a numerical systems called “factoradics” and describe an algorithm based on this point of view. We analyse this algorithm using tools from analytic combinatorics, which actually give a systematic way to study all the algorithms form the literature in a unified framework. Finally, in Section 3.2.3, we demonstrate experimentally that the classical complexity model at use to analyse such algorithms is not realistic enough as it underestimates the cost of large integer arithmetic. In light of this, we revise our implementation to minimise the cost of arithmetic operations and propose a refined complexity analysis. The outcome of this section is a significantly faster algorithm than the historical ones, whose experimental runtime fits the complexity established in our refined complexity model.

3.2.1 Historical context and classical results

The problem of *unranking* combinations dates back to the 1970s but some closely related questions were studied before. Several algorithms were developed to solve this problem, which re-

volve around two main ideas: the recursive method and the use of a special numeral system called "combinadics".

Unranking via the recursive method

As we explained in the introduction of the present chapter, one natural solution to generate or unrank a combinatorial object is to rely on a recursive decomposition. If $\mathcal{A}_{n,k}$ denotes the set of the combinations of k elements among n, then we have that $\mathcal{A}_{n,k} = \mathcal{A}_{n-1,k-1} \times \{n-1\} + \mathcal{A}_{n-1,k}$. This decomposition naturally allows to unrank combinations in *reverse co-lexicographic* order (denoted $\prec_{\text{rev colex}}$). This order is similar to the lexicographic order ($\prec_{\text{lex}}$) but the combinations are read from right to left (reverse) and the order on the integers is used backwards (co-lexicographic). This is formally defined as follows:

$$(u_i)_{1\le i\le k} \prec_{\text{rev colex}} (v_i)_{1\le i\le k} \iff (n-1-u_{k+1-i})_{1\le i\le k} \prec_{\text{lex}} (n-1-v_{k+1-i})_{1\le i\le k}.$$

All the combinations from $\mathcal{A}_{n-1,k-1} \times \{n-1\}$ are smaller than those from $\mathcal{A}_{n-1,k}$ for $\prec_{\text{rev colex}}$. Hence, a lexicographic unranking algorithm is obtained by first unranking a combination in reverse co-lexicographic order using the straightforward algorithm from [NW78], and then applying the transformation $(u_i)_{1\le i\le k} \mapsto (n-1-u_{k+1-i})_{1\le i\le k}$.

The general principles to write such an algorithm are given in [NW78], an iterative implementation is given in [Rus03, page 65] and we provide a recursive implementation in [GP21]. We only explain the reverse-lexicographic *core* of the algorithm in Algorithm 32. The idea of the algorithm is simple in this case as we only use the "disjoint union" rule of the recursive method. The base cases of the algorithm are those where $k = 0$, in which case the only combination is the empty sequence, and $k = n$, in which case the only combination is the complete sequence $(i)_{0\le i<n}$. In the general case, if the rank u is smaller than the number of elements of $\mathcal{A}_{n-1,k-1} \times \{n-1\}$, that is $\binom{n-1}{k-1}$, then we unrank the u-th element of this set. Otherwise, "skip" the elements of $\mathcal{A}_{n-1,k-1} \times \{n-1\}$ by subtracting $\binom{n-1}{k-1}$ from the rank and we unrank an element of $\mathcal{A}_{n-1,k}$.

Algorithm 32 Reverse co-lexicographic unranking of combinations (recursive method)

Input: Three integers n, k and u such that $0 \le k \le n$ and $0 \le u < \binom{n}{k}$
Output: The u-th combination of k among n elements in reverse co-lexicographic order

```
function RevCoLexUnrank(n, k, u)
    if k = 0 then return the empty sequence
    else if k = n then return (0, 1, 2, ..., n − 1)
    else if u < binom(n−1, k−1) then
        C ⟵ RevCoLexUnrank(n − 1, k − 1, u)
        add n − 1 at the end of C
        return C
    else return RevCoLexUnrank(n − 1, k, u − binom(n−1, k−1))
```

Unranking via combinadics

An alternative and rather different approach comes from Lehmer who studied the inverse problem of *ranking* in [Pól+64, page 27]. Given a combination, what is its lexicographic rank? Lehmer's approach is based on a numeral system called "combinatorial number system" or "combinadics", which had already appeared in the literature in 1887 in [Pas87] and allows to decompose integers as sums of binomial coefficients. More precisely, given an integer $0 \le u < \binom{n}{k}$, there exists a unique sequence $0 \le c_1 < c_2 < \cdots c_k < n$ such that

$$u = \binom{c_1}{1} + \binom{c_2}{2} + \cdots + \binom{c_{k-1}}{k-1} + \binom{c_k}{k}. \tag{3.4}$$

The sequence $(c_1, c_2, \dots, c_k)$ is called the *combinadic* of u.

This decomposition describes a bijective mapping between the interval $[\![0; \binom{n}{k} - 1]\!]$ and the set of the increasing sequences of k elements from the set $[\![0; n-1]\!]$ or, put differently, the set of combinations. Lehmer used this bijection to develop a ranking algorithm and an algorithm implementing the reverse mapping was published later in 1977 in [BL77]. This is very close to the algorithm used in Matlab [But15]. Alternative algorithms, also based on the combinadic decomposition of the rank, were published later, notably [Er85] and [KS99, page 47] (also appearing later in the MSDN article [McC04]).

For the sake of conciseness, we only recall the algorithm from [BL77] in Algorithm 33. The complete list of the algorithms mentioned here is given and analysed in more detail in our paper [GP21]. The idea of Algorithm 33 is that for decomposing u in combinadics as in (3.4), one must find the largest m such that $\binom{m}{k} \le u$. By definition, we then have that $u < \binom{m+1}{k}$ and, as a consequence, $u' = u - \binom{m}{k} < \binom{m}{k-1}$. Hence, we can decompose u' in combinadics as an integer of $[\![0; \binom{m}{k} - 1]\!]$ so that $u' = \sum_{i=1}^{k-1} \binom{c_i}{i}$ and we conclude by setting $c_k = m$. The while loop in Algorithm 33 implements the search for the right value of m.

Algorithm 33 [BL77]'s algorithm for unranking combinations (combinadic approach)

Input: Three integers n, k and u such that $0 \le k \le n$ and $0 \le u < \binom{n}{k}$
Output: The u-th combination of k among n elements in lexicographic order

```
function UnrankViaCombinadics(n, k, u)
    L ← [0, ..., 0] (k components),    r ← 0
    for i from 0 to k − 2 do
        if i ≠ 0 then L[i] ← L[i − 1] else L[i] ← −1
        while true do
            L[i] ← L[i] + 1
            b ← binom(n − L[i] − 1, k − i − 1)
            r ← r + b
            if r > u then exit the loop
        r ← r − b
    L[k − 1] ← L[k − 2] + u − r + 1
    return L
```

Complexity results

All the algorithms presented above need to compute binomial coefficients during their execution. The usual way to evaluate the efficiency of such algorithms is to count the number of times the function computing binomial coefficients is called (see [Rus03, page 66], [BL77] and [Er85] for instance). In this complexity model, all the aforementioned algorithms have been proven to be linear in average. More precisely, given $0 \le k \le n$, for all these algorithms, the average (over all the possible ranks $0 \le u < \binom{n}{k}$) of the number of binomial coefficients computed during an execution is of the form $n + O(1)$ where the $O(1)$ error term is uniform in k. Note that this is also the worst case.

Although this seems satisfactory, we will see in Section 3.2.3 that this complexity model does not reflect the actual runtime of these algorithms. We will then provide a more realistic complexity analysis and show that there is still room for improvement in these algorithms.

3.2.2 Factoradics

We present here an third approach to the unranking problem, using a different numeral system called factorial number system or "factoradic". While the term "factoradic" is more recent, this system already appears in the literature at the end of the 19-th century in [Lai88] and is also mentioned in [Knu97, page 209].

Definition 18. *Let $u \ge 0$ and let n be the unique non-negative integer satisfying $(n-1)! \le u < n!$ (adopting the convention than $(-1)! = 0$). Then, there exists a unique sequence $(f_\ell)_{0 \le \ell < n}$ such that $0 \le f_\ell \le \ell$ for all ℓ and*

$$f_0 \cdot 0! + f_1 \cdot 1! + f_2 \cdot 2! + \cdots + f_{n-1} \cdot (n-1)! = u.$$

The finite sequence $(f_0, f_1, \dots, f_n)$ is called the factoradic *decomposition of u (or just* factoradic *of u for short).*

Just like combinadics are linked with combinations, factoradics are closely related to permutations. Indeed, the factoradic decomposition $(f_1, f_2, \dots, f_n)$ of an integer can be converted into a permutation by using the ideas from the Fisher–Yates algorithm [FY48; Dur64]. It consists in selecting the first element σ_1 of the permutation uniformly at random in the set $[\![0; n-1]\!]$, then selecting the second element σ_2 uniformly at random in $[\![0; n-1]\!] \setminus \{\sigma_1\}$, then the third one in $[\![0; n-1]\!] \setminus \{\sigma_1, \sigma_2\}$, etc. Here, the idea is to use the factoradic decomposition to choose which element to select rather than selecting it at random. This is described in Algorithm 34.

The idea to unrank combinations using factoradics is to encode them as permutations. Then, unranking the u-th combination of k elements among n consists in (1) converting the combination rank u into the permutation rank of the corresponding permutation and (2) unranking this permutation. The encoding is defined below.

Definition 19. *Let n and k be two integers such that $0 \le k \le n$. We define $\mathcal{P}_{n,k}$ as the function which maps the combination $(\ell_0, \ell_1, \dots, \ell_{k-1}) \in \mathcal{A}_{n,k}$ to the size-n permutation $(\ell_0, \ell_1, \dots, \ell_{k-1}, d_k, \dots, d_{n-1})$ where the integers d_i appear in increasing order and where $\{\ell_0, \ell_1, \dots, \ell_{k-1}, d_k, \dots, d_{n-1}\} = [\![0; n-1]\!]$.*

Algorithm 34 Lexicographic unranking of permutations based on factoradics

Input: Two integers $n \leq 0$ and u such that $0 \leq u < n!$
Output: The u-th permutation in lexicographic order, as an array
function UNRANKPERM(n, u)
  $(f_\ell)_{0 \leq \ell \leq n} \longleftarrow$ factoradic decomposition of u (padded with zeros at the end if necessary)
  $S \longleftarrow \{0, 1, 2, \dots, n-2, n-1\}$
  $P \longleftarrow [0, 0, 0, \dots, 0, 0]$ (n components)
  for i **from** 0 **to** $n-1$ **do**
    $P[i] \longleftarrow$ the (f_{n-i})-th element of S
    Remove $P[i]$ from S
  return P

In order to express the conversion from combination rank to permutation rank, we first characterise the factoradic decompositions which form the image of $\mathcal{P}_{n,k}$.

Lemma 8. *Let n and k be two integers such that $0 \leq k \leq n$. The function $\mathcal{P}_{n,k}$ is a bijection from $\mathcal{A}_{n,k}$ to the set of size-n permutations whose prefix of length k and suffix of length $n-k$ are both increasingly sorted. Moreover, the permutations in the image of $\mathcal{P}_{n,k}$ have a factoradic decomposition of the form $(0, 0, \dots, 0, f_{n-k}, f_{n-k+1}, \dots, f_{n-1})$ where $n-k \geq f_{n-k} \geq f_{n-k+1} \geq \cdots \geq f_{n-1} \geq 0$.*

Proof. The function $\mathcal{P}_{n,k}$ is injective since the length k prefix of the image of a combination is the combination in question. Moreover, there is only one way to complete this prefix with the remaining values of $[\![0; n-1]\!]$ in increasing order. Hence, $\mathcal{P}_{n,k}$ is a bijection.

Now, let u be an integer whose factoradic is $(0, \dots, 0, f_{n-k}, \dots f_{n-1})$. Due to the constraint $n-k \geq f_{n-k} \geq f_{n-k+1} \geq \cdots \geq f_{n-1} \geq 0$, the permutation corresponding to u has for prefix of length k the sequence $(f_{n-1}, f_{n-2}+1, f_{n-3}+2, \dots, f_{n-k}+k-1)$ which is increasingly sorted. Moreover, since the remaining indices of the factoradic are only zeros, the rest of the permutation (the suffix of length $n-k$) corresponds to the increasing sequence of elements of $[\![0; n-1]\!]$ that have not yet been taken. Hence, the permutation encoded by u corresponds to a combination via $\mathcal{P}_{n,k}^{-1}$.

Conversely, let $(\ell_0, \ell_1, \dots, \ell_{k-1}) \in \mathcal{A}_{n,k}$ be a combination. The factoradic of its image by $\mathcal{P}_{n,k}$ is given by $(0, 0, \dots, 0, \ell_{k-1}-(k-1), \ell_{k-2}-(k-2), \dots, \ell_0)$, which satisfies the conditions of the lemma. □

Before giving a first version of our unranking algorithm, note that the proof of Lemma 8 gives us a direct way to unrank a permutation in the image of $\mathcal{P}_{n,k}$, once its factoradic is known. Indeed, we have an explicit formula for the prefix of length k of the permutation whose factoradic is $(0, \dots, 0, f_{n-k}, f_{n-k+1}, \dots, f_{n-1})$. In addition, for the sole purpose of unranking a combination, we do not need to compute the $(n-k)$ remaining values of the permutation as they are discarded by $\mathcal{P}_{n,k}^{-1}$. As a consequence, the only difficult part of our unranking algorithm is the conversion from combination rank to permutation rank and the decomposition in factoradic.

Since the components of the factoradic decomposition of a partition are read from right to left, the *lexicographic* unranking of combinations is achieved by first considering the sequences

of the form $(0, \dots, 0, f_{n-k}, f_{n-k+1}, \dots, f_{n-1})$ such that $f_{n-1} = 0$, then those such that $f_{n-1} = 1$, etc. To do this efficiently, we use the following combinatorial argument.

Lemma 9. *For any non-negative integers m, n and k, the number of sequences $(f_i)_{0 \le i < k}$ satisfying $n \ge f_0 \ge f_1 \ge \dots \ge f_{k-1} \ge m$ is given by $\binom{n-m+k}{k}$.*

Proof. Such a sequence can be encoded as a word of length $(n - m + k)$ on the alphabet $\{a, b\}$ with exactly k occurrences of b where the i-th occurrence of b $(1 \le i \le k)$ encodes the value of f_{n-i} as m plus the number of occurrences of a preceding this b. For instance, for $m = 0$, $n = 5$ and $k = 4$, the sequence $(4, 2, 1, 1)$ is encoded by the word $abbabaaba$. All such words correspond to a unique sequence and their number is given by $\binom{n-m+k}{k}$. □

Using Lemma 9, we can for instance get that the number of factoradic decompositions corresponding to combinations of $\mathcal{A}_{n,k}$ and such that $f_{n-1} = m$ is given by $\binom{n-m-1}{k-1}$. We use this argument to recursively determine all the f_{n-i} in Algorithm 35.

Algorithm 35 Lexicographic unranking of combinations using the factoradic approach

Input: Three integers n, k and u such that $0 \le u < \binom{n}{k}$
Output: The u-th combination of k elements among n in lexicographic order

```
function CombUnrankFact(n, k, u)[1]
    C ⟵ [0, ..., 0] (k components),   i ⟵ 0,   m ⟵ 0
    while i < k do
        b ⟵ binom(n−1−m−i, k−1−i)
        if u < b then
            C[i] ⟵ m + i
            i ⟵ i + 1
        else
            u ⟵ u − b
            m ⟵ m + 1
    return C
```

The correction of Algorithm 35 is ensured by Lemma 9 and the above discussion. We first assess the efficiency of Algorithm 35 in terms of binomial coefficient evaluations, which is equivalent to count the number of iterations of the while loop. Since the value of $(m + i)$ increases at each iteration and can be at most $(n - 1)$, the worst case complexity of the algorithm is n and is obtained for all the combinations whose maximum element is $(n-1)$. In Theorem 20 we establish the *average* complexity of this procedure.

Theorem 20. *Let $u_{n,k}$ denote the* cumulative *number of binomial coefficients evaluated while unranking* all *the combinations of k elements among n using Algorithm 35. We have that*

$$u_{n,k} = \binom{n}{k} \frac{k}{k+1}(n+1).$$

Proof. We first establish a recurrence formula satisfied by $u_{n,k}$ by reasoning on the first iterations of the while loop. First note that if $k = 0$ or if $k > n$, then $u_{n,k} = 0$. For all the other values of k, one coefficient is evaluated upon entering the while loop and depending of the outcome of the comparison at line 5, two things can happen.

- If $u < b$, then the first component of the combination is 0 and the rest of the execution of the algorithm boils down to unranking a combination of $(k-1)$ elements among the $(n-1)$ elements of $[\![1; n-1]\!]$. The cumulated cost of the rest of the evaluation for all the ranks $u < b$ is thus $u_{n-1,k-1}$.
- Conversely, if $u \geq b$, then then smallest elements of the combinations is not 0 and the rest of the evaluation boils down to unranking a combination of k elements among the $(n-1)$ elements of $[\![1; n-1]\!]$. The cumulative cost of the rest of the evaluation for all those ranks is thus $u_{n-1,k}$.

As a consequence, the sequence $u_{n,k}$ satisfies $u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}$ where the first terms accounts for the systematic evaluation of a binomial coefficient for all the ranks $0 \leq u < \binom{n}{k}$.

Let $U(z, y)$ be the generating function associated with the sequence $(u_{n,k})$, that is $U(z, y) = \sum_{n,k\geq 0} u_{n,k} z^n y^k$. Using the recurrence relation we have just established, we have that

$$U(z, y) = \sum_{0<k\leq n} \binom{n}{k} z^n y^k + zyU(z, y) + zU(z, y) = \frac{1}{1-(1+y)z} B(z, y)$$

where

$$B(z, y) = \sum_{0<k\leq n} \binom{n}{k} z^n y^k = \sum_{n>0} ((1+y)^n - 1) z^n = \frac{1}{1-(1+y)z} - \frac{1}{1-z}.$$

The last step of the proof consists in extracting the coefficient $u_{n,k}$ in the expansion of $U(z, y)$. First we extract the coefficient in front of z^n:

$$\begin{aligned}[z^n]U(z, y) &= \sum_{\ell=0}^{n-1} [z^\ell] \frac{1}{1-(1+y)z} \cdot [z^{n-\ell}] B(z, y) = \sum_{\ell=0}^{n-1} (1+y)^\ell \cdot ((1+y)^{n-\ell} - 1) \\ &= n(1+y)^n - \frac{1-(1+y)^n}{y}.\end{aligned}$$

This equality gives the distribution of the costs as a function of $k \in [\![0; n]\!]$ and finally, the value of $u_{n,k}$ is obtained by extracting the coefficient in front of y^k:

$$[z^n y^k]U(z, y) = n\binom{n}{k} - \binom{n}{k+1} = \binom{n}{k} \frac{k}{k+1}(n+1). \qquad \square$$

A consequence of Theorem 20 is that the average cost of unranking a combination is given by $u_{n,k}\binom{n}{k}^{-1} = \frac{k}{k+1}(n+1) = \frac{k(n+1)}{(k+1)n} n$, which is close to the worst case n. In particular as soon as $k \longrightarrow \infty$ we have:

$$\frac{u_{n,k}}{\binom{n}{k}} \underset{\substack{n\to\infty \\ k\to\infty}}{=} n + o(n).$$

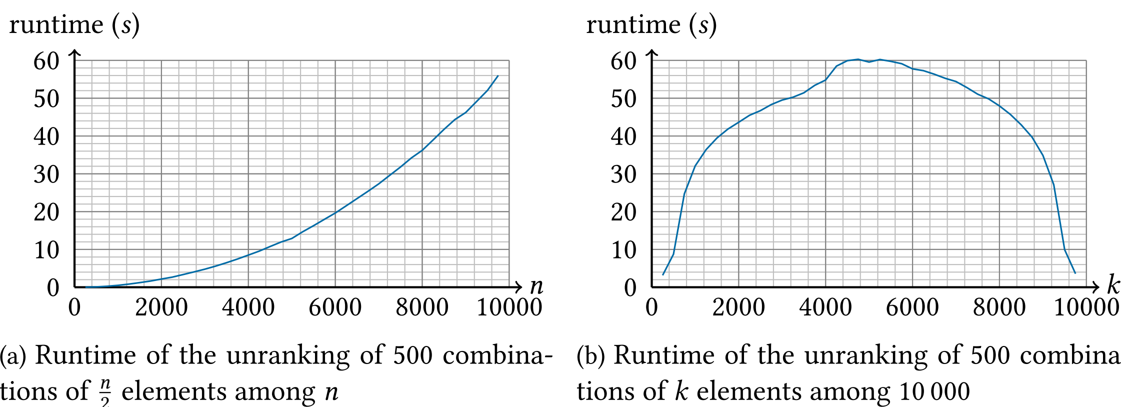
Our algorithm is thus on par with the existing ones mentioned in Section 3.2.1 and simply provides an alternative point of view on the problem. However, we will now see that the complexity model used here is not realistic enough and provide a finer complexity analysis.

3.2.3 Realistic complexity analysis and algorithmic improvements

Here we challenge the assumption that computing a binomial coefficient can be considered to be of unit cost. This may be true if one limits itself to bounded integers or if n and k are small enough so that all the binomial coefficients can be pre-computed and stored in memory, but this assumption fails for large values of n. We highlight this using two experiments.

First, we measure the runtime of unranking combinations of $k = \frac{n}{2}$ elements among n when n varies. More precisely, we measure the total runtime of the unranking algorithm for 500 input ranks drawn uniformly at random in $[\![0; \binom{n}{k} - 1]\!]$, for each pair (n, k), and where n is ranging from 250 to 10 000 by steps of 250 and $k = n/2$. The results are presented in Figure 3.1a. Expectedly, this shows that the runtime of Algorithm 35 is super-linear which suggests that the cost of computing one binomial coefficient grows with n.

Then, in a second experiment, we fix $n = 10\,000$, and we measure the total runtime of unranking 500 uniform combinations for each value of k ranging from 250 to 9 750 by steps of 250. The results of this experiment are presented in Figure 3.1b. This shows a rather different evolution from what one could expect from the complexity result exposed above. In fact the experimental worst case (in terms of runtime) seems to be at $k = \frac{n}{2}$ unlike the theoretical worst case (in terms of binomial coefficients evaluations) which is at $k = n$. Moreover, the complexity result presented above only showed a small dependence in k. Here we see that the runtime of the algorithm is significantly higher for $k = \frac{n}{2}$ than for k close to 0 or close to n.



(a) Runtime of the unranking of 500 combinations of $\frac{n}{2}$ elements among n

(b) Runtime of the unranking of 500 combinations of k elements among 10 000

Figure 3.1: Experimental study of the naive unranking algorithm

Avoiding the computation of binomial coefficients

The complexity of computing one binomial coefficient is actually a complicated question. One common choice is to exploit the formula $\binom{n}{k} = \binom{n-1}{k-1}\frac{n}{k}$ to compute $\binom{n}{k}$ recursively in only k multiplications and k divisions of a large integer (a binomial coefficient) by a small integer (smaller than n). An asymptotically faster approach consists in computing the product $n(n-1)\cdots(n-k+1)$ and the factorial $k!$ independently using a divide and conquer approach similar to [Bos+17, page 275] and to compute the binomial coefficient using the formula $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!}$. In the experiment from Figure 3.1 we used the GMP [7] library for the computation of the binomial

coefficients, which implements several algorithms and tries to choose the best one depending on the values of n and k.

Here, in the light of our experimental study, we propose an alternative unranking algorithm which avoid as much as possible the computation of binomial coefficients "from scratch" by re-using previously computed values. More precisely, every time a new binomial coefficient is needed, instead of forgetting the last computed one and calling a dedicated procedure to compute the new one, we use the two following identities to compute the new one from the last one:

$$\binom{n}{k} = \binom{n-1}{k-1}\frac{n}{k} \qquad\qquad \binom{n}{k} = \binom{n-1}{k}\frac{n}{n-k}.$$

This is possible and efficient thanks to the fact that from one iteration of the loop of Algorithm 35 to the next one the values of $(n-1-m-i)$ and $(k-i)$ change by at most 1. As an example, at the very first iteration of the loop, the variable b holds the value $\binom{n-1}{k-1}$. If the condition $u < b$ is satisfied i is incremented by one and at the next iteration b will hold the value $\binom{n-2}{k-2}$. Rather than calling the procedure computing the binomial coefficient from scratch, the new value of b can be computed by the instruction $b \longleftarrow (b \cdot (k-1)) / (n-1)$. Similarly, if the condition $u < b$ is not satisfied at the first iteration, then m gets incremented by 1 and b will hold the value $\binom{n-2}{k-1}$ at the next iteration. This value can be obtained by $b \longleftarrow (b \cdot (n-k)) / (n-1)$. The new algorithm, exploiting this trick to accelerate the unranking, is given in Algorithm 36. Note that in addition to using this trick, we also save a few computations by deducing the value of the last component of the combination directly from the rank at line 13.

Algorithm 36 Optimised lexicographic unranking of combinations using the factoradic approach

Input: Three integers n, k and u such that $0 \le u < \binom{n}{k}$
Output: The u-th combination of k elements among n in lexicographic order

```
1:  function CombUnrankFactOptimised(n, k, u)[1]
2:      C ⟵ [0, ..., 0] (k components),   i ⟵ 0,   m ⟵ 0
3:      b ⟵ binom(n−1, k−1)
4:      while i < k − 1 do                      ▷ Loop invariant: b = binom(n−m−i−1, k−i−1)
5:          if u < b then
6:              C[i] ⟵ m + i
7:              b ⟵ b · (k − i − 1) / (n − m − i − 1)
8:              i ⟵ i + 1
9:          else
10:             u ⟵ u − b
11:             b ⟵ b · (n − m − k) / (n − m − i − 1)
12:             m ⟵ m + 1
13:     if k > 0 then C[k − 1] ⟵ n + u − b · (n − m − k + 1)
14:     return C
```

Complexity analysis

The improved version of our unranking algorithm presented in Algorithm 36 allows to reduce the cost of computing one binomial coefficient to one multiplication and one division of a large integer by a small integer. Here we quantify this cost precisely by counting the number of bitwise operations spent in arithmetic operations by the algorithm. To this end, we re-use the earlier result from Theorem 20 which states that both in average and in the worst case, about n iterations of the loop are performed. Here we give an upper bound on the cost of the arithmetic operations at lines 7 and 11 in Algorithm 36.

Lemma 10. *Let $\alpha \in \mathbb{R}$ such that $0 < \alpha < 1$. If $n \longrightarrow \infty$ and $k \sim \alpha n$, then we have*

$$\log_2 \binom{n}{k} \sim n \cdot L(\alpha) \quad \textit{where} \quad L(\alpha) = (1-\alpha)\log_2 \frac{1}{1-\alpha} + \alpha \log_2 \frac{1}{\alpha}\cdot$$

Proof. By Stirling's formula, we have that $\log_2(n!) = n\log_2(n) + O(n)$, hence

$$\begin{aligned}
\log_2(k!) &= k\log_2(k) + O(k) = \alpha n \log_2(\alpha n) + o(n\log_2(n)) \\
\log_2((n-k)!) &= (1-\alpha)n\log_2((1-\alpha)n) + o(n\log_2(n)) \\
\log_2 \binom{n}{k} &= \log_2(n!) - \log_2(k!) - \log_2((n-k)!) \\
&\sim n \cdot \Big((\alpha + (1-\alpha))\log_2(n) - \alpha\log_2(\alpha n) - (1-\alpha)\log_2((1-\alpha)n) \Big) \\
&\sim n \cdot \Big((1-\alpha)\log_2 \frac{1}{1-\alpha} + \alpha\log_2 \frac{1}{\alpha} \Big).
\end{aligned}$$

□

Finally, we can use the fact that all the binomial coefficients used in the algorithm are upper-bounded by $\binom{n}{k}$ and combine Lemma 10 with Lemma 1 on page 6 from the "Generalities" chapter. This yields an upper bound on the bit-complexity of Algorithm 36.

Theorem 21. *The number of bitwise operations spent in arithmetic computations in Algorithm 36, when called with $k \sim \alpha n$, admits an upper bound of the form*

$$C \cdot L(\alpha) \cdot \frac{n^2}{\log_2(n)} \cdot \mathcal{M}(\log_2(n))$$

for some constant C independent from α and where $\mathcal{M}(s)$ is the cost of one multiplication[3] *of two integers of bit-size at most s.*

Proof. All the binomial coefficients manipulated by the algorithm are upper-bounded by $\binom{n}{k}$. Moreover, by Lemma 1 on page 6, the cost of one multiplication of a number of bit-size at most $\log_2(n)$ by a number of bit-size at most $nL(\alpha)$ is at most $\frac{nL(\alpha)}{\log_2(n)}\mathcal{M}(\log_2(n)) + O(n/\log_2(n))$. Finally the algorithm performs a linear number of operations in the worst case (and in average). Also note that the cost of computing the initial binomial coefficient at line 3 is negligible compared to n^2. □

[3] See the "generalities" Chapter on page 6

Note that the presence of the term $\mathcal{M}(\log_2(n))$ in the statement of the theorem corresponds to multiplications of small integers (bounded by n), which we must take into account in our complexity model. However, in practice these terms will always fit in a machine word so that the "practical" cost of the multiplication of a binomial coefficient by a small integer is in fact of the order of $L(\alpha)n$ operations on *machine words*. This implies the runtime one should observe in practice should behave like $O(n^2)$.

In order to validate our result, we compare the experimental runtime of Algorithm 36 for a fixed n when k varies from 0 to n, with the theoretical upper-bound of the form $k \mapsto C_n \cdot L(k/n)$ established in Theorem 21. The constant C_n has been chosen numerically so that the two curves coincide at $k = n/2$. The results of this experiment are give in Figure 3.2. We can see on this plot that Theorem 21 reflects faithfully the actual runtime of the algorithm. Besides, we can observe that the algorithm runs significantly faster than the previous version without the optimisation related to binomial coefficients.

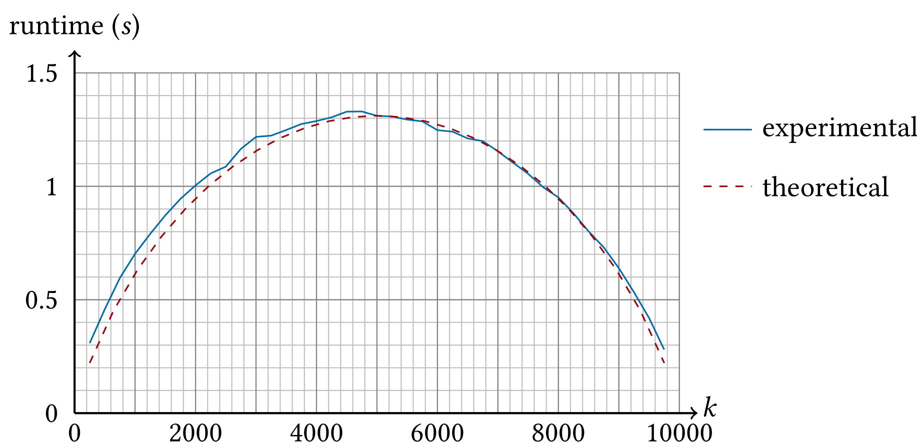


Figure 3.2: Comparison between the runtime of the unranking of 500 combinations of k elements among 10 000 and the theoretical complexity established in Theorem 21

As a final remark in this section, we would like to highlight that the optimisation we described here is not only applicable to the specific problem of unranking combinations but also to many other algorithms manipulating big integers. As an example, in Algorithm 9 on page 43 in Chapter 1, we use the exact same trick for the computation of the coloured product of two polynomials. Similarly, the complexity results regarding the enumeration of DOAGs in Chapter 2 (Theorem 12 on page 65) relies on a similar technique being used so that the evaluation of the binomial coefficients and factorials in the sum are negligible. Finally, a more sophisticated application of this principle is also available in the next section in Algorithm 40 on page 117. There, no binomial coefficients are involved but the same idea applies: if a recursive formula is available to re-use computations from the last iteration of the loop, we can save computation time.

3.3 Uniform random sampling of variations

A basic, though fundamental combinatorial class is the class of variations. In this section, we briefly recall the notion of variation and we describe an efficient uniform random sampler and an unranking procedure for this class.

3.3.1 Definition and enumeration

Definition 20 (Variation)**.** *We call a* variation *of size n a finite sequence of n non-negative integers such that:*

- *the non-zero integers may occur at most once in the sequence;*
- *if $i > 0$ occurs in the sequence, then for all $0 < j < i$, j occurs in the sequence.*

A variation is said to be non-empty if there is at least one positive number is the sequence.

For instance, the sequence $v = (6, 2, 3, 0, 0, 1, 4, 0, 5)$ is a variation of size 9. In this section, we only consider non-empty variations and from now on we refer to them as "variations". One of the earliest references to these objects dates back to 1659 in Izquierdo's *Pharus scientiarum* [Izq59, Disputatio 29]. They also appear in Stanley's book in the *Twelvefold Way* [Sta86, page 31], a collection of twelve basic counting problems arising frequently in practice. In our context, the random generation of variations is a key component of our rejection-based random sampler of DOAGs in Section 2.2.5.

Another way to present variations is to describe them as the interleaving of a permutation with a sequence of zeros. As a consequence, the combinatorial class of variations $\mathcal{V}$ admits the following labelled specification

$$\mathcal{V} = \textsc{Seq}_{\geq 1}(\mathcal{Z}) \star \textsc{Set}(\mathcal{Z}).$$

In this specification, the labelled sequence starting from 1 represents a non-empty permutation and the labelled set represents the zeros of the variation. This corresponds to an encoding of variations where the zeros are replaced by an increasing sequence of integers, all larger than the integers used for the permutation. We will re-use this encoding later in this section to describe our uniform random sampler.

The exponential generating function of variations is thus given by $\frac{z}{1-z}\exp(z)$ and a formula for the number v_n of variations of size n can be obtained by extracting the n-th coefficient of this series. Hence, we have

$$v_n = n![z^n]\frac{z}{1-z}\exp(z) = n!\sum_{p=0}^{n}[z^{n-p}]\frac{z}{1-z}[z^p]\exp(z) = n!\sum_{p=0}^{n-1}\frac{1}{p!} = e \cdot n! - 1 - O\left(\frac{1}{n}\right). \quad (3.5)$$

This sequence also has a simple recurrence formula allowing to efficiently compute the n first terms of the sequence. For all $n > 0$, we have $v_n = n \cdot (v_{n-1} + 1)$. The combinatorial interpretation behind this formula is that a variation of size n can be decomposed as:

- the position j of its largest element in the sequence;
- and the sequence of integers obtained by removing its j-th element, which can be either a variation of size $(n-1)$ or a sequence of zeros.

Using this recurrence, we computed the first terms of v_n which corresponds to the sequence stored under the reference A007526 in the OEIS. The 10 first terms of the sequence are: 1, 4, 15, 64, 325, 1956, 13699, 109600, 986409, 9864100.

The algorithm computing the n first terms of the sequence v_n has to deal with big-integer arithmetic since $v_n \sim e \cdot n!$ diverges quickly and hence only fits in a machine word for small values of n. A good measure of complexity for such algorithms is given by the number of bitwise operations rather than the number of arithmetic operations on integers. In this model, the complexity of the addition of two numbers with at most n bits is linear in n and we denote by $\mathcal{M}(n)$ the complexity of one multiplication (see the generalities chapter at page 6).

Theorem 22 (Counting)**.** *Computing the values of v_k for all $k \le n$ can be achieved in complexity $O(n^2\mathcal{M}(\log_2(n)))$ in terms of bitwise operations. Moreover, storing the binary representations of these numbers requires $\Theta(n^2 \log_2(n))$ bits in memory.*

Proof. We have that $\log_2(v_k) \sim k\log_2(k)$, hence the number $(v_{k-1}+1)$ has a bit-size of $O(k\log_2(k))$ and k has a bit-size of $O(\log_2(k))$. By Lemma 1 on page 6, the cost of the multiplication of $(v_{k-1}+1)$ by k is thus in $O(k\mathcal{M}(\log_2(k)))$. Hence, the cumulated cost of computing all these numbers up to $k = n$ is $O(n^2\mathcal{M}(\log_2(n)))$. Finally, since the bit-size of v_n is of the order of $n\log_2(n)$, the total space required to store all these numbers is of the order of $\sum_{k=1}^{n} k\log_2(k) = \Theta(n^2\log_2(n))$. □

Note that in practice n is likely to fit in a machine word so that the multiplications accounted for by $\mathcal{M}(\log_2(k))$ are simple multiplications on machine words and the overall procedure only performs $n^2\log_2(n)$ such operations. To give a rough idea of the performance of this algorithm, its straightforward implementation in C using the GMP library for big-integer arithmetic is able to compute the $n = 50000$ first terms of the sequence in about a second.

3.3.2 Random sampling

The recursive method

A natural approach to sample objects that have a recursive decomposition as described above is to resort to the recursive method explained above in Section 3.1. In our case, in order to sample a uniform variation of size n, the idea is to:

1. first draw the position of the largest element of the sequence uniformly at random (this corresponds to the factor n in the recursive formula);
2. and then either fill the rest of the sequence with zeros or with a uniform variation of size $(n-1)$ (this is accounted for by $(v_{n-1}+1)$). The decision to stop the generation and filling the rest of the sequence with zeros is based on a Bernoulli variable of parameter $\frac{1}{1+v_{n-1}}$,

This procedure is given in Algorithm 37. It assumes that the numbers v_n have been pre-computed or are computed lazily using a dynamic programming approach.

In order to implement the insertion of $(j+1)$ at position p as described at line 6 in the algorithm, it is advisable not to use a array to represent the variations. For instance, a more efficient representation is to use a binary tree constructed recursively as follows.

Algorithm 37 Uniform random sampler of variations using the recursive method

Input: An integer $n > 0$
Output: A uniform random variation of size n

```
1:  function UnifVariationRec(n)
2:      p ← Unif([[0; n − 1]])
3:      if Bern(1/(1+v_{n−1})) then
4:          v ← UnifVariationRec(n − 1)
5:          j ← maximum element of v
6:          insert (j + 1) at position of the p-th zero of v
7:      else
8:          v ← [0, 0, ..., 0]                              ▷ length n
9:          v[p] ← 1
10:     return v
```

- The root stores the smallest non-zero element ℓ of the array, the number of zeros of the array, a sub-tree constructed recursively from the element on the left of ℓ in the array, and a second sub-tree constructed recursively from the element on the right of ℓ in the array.
- If an array only contains zeros, then it is represented by a leaf only storing the length of the array.

Inserting an element at the position of the p-th zero in this data-structure is similar to inserting an element in a binary search tree and is thus logarithmic in average. The insertion of an element in the tree representation of an example variation is given in Figure 3.3.

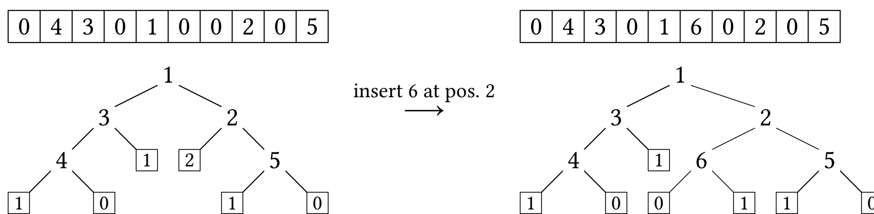


Figure 3.3: Insertion of the element 6 at position 2 (counting from 0) in an example variation based on its binary tree representation. The leaves of the trees are represented as squares and the number of zeroes they encode in written inside the squares.

Theorem 23 (Complexity of Algorithm 37)**.** *Algorithm 37 can be implemented so as to perform $O(n\log(n))$ memory accesses and to consume $O(n\log(n))$ random bits, in expectation.*

Proof. The $O(n\log(n))$ memory accesses can be achieved using the memory layout discussed above for the remaining indices. The $O(n\log(n))$ bound for the number of random bits consumed comes from the fact that each call to the Unif function consumes at most $\lceil\log(n)\rceil$ bits and generating each Bernoulli variable account for a constant number of bits in expectation. □

The major drawback of this algorithm is that it requires the pre-computation of a large number of big integers. This requires a significant amount of space and has a super-quadratic cost on

7	4	1	0	3	6	0	0	5	2

$\longrightarrow$

7	4	1	**8**	3	6	**9**	**10**	5	2

$p = 3$

Figure 3.4: A variation of size 10 and its corresponding pair (σ, p) of a permutation and an integer by Proposition 5.

the computation time. In the following, we describe a better algorithm which does not require any pre-computation and is linear in expectation.

Rejection-based sampling

Another strategy for generating combinatorial objects uniformly at random in a given class is to sample uniform objects in a super-set of the targeted class until a valid object is obtained. This is particularly relevant when the super-set in question is both easily amenable to uniform random sampling and not too big compared to the targeted class. This technique is called the *rejection* principle and has been introduced in Section 3.1. To this end, we first present an alternative representation of variations in Proposition 5. An example variation of size 10 and its alternative representation are pictured in Figure 3.4.

Proposition 5. *The set of non-empty variations of size n is in bijection with the set of pairs (σ, p) of a permutation σ of size n and of an integer $p \in [\![0; n-1]\!]$ such that the p largest elements of σ appear in increasing order in σ.*

Proof. Given a non-empty variation V, we define p as the number of zeros of V and σ as the permutation obtained by replacing the zeros of V by the numbers $n-p+1, n-p+2, \ldots, n$ in increasing order.

Conversely, given a pair (σ, p) as described in the statement of the theorem, we define its corresponding variation as the sequence obtained by replacing the p largest values of σ by 0. □

Using this representation, the class $\mathcal{V}_n$ of the variations of size n can be seen as a sub-set of $\mathfrak{S}_n \times [\![0; n-1]\!]$. This latter set is a good candidate for rejection-based sampling since sampling one of its elements uniformly at random is computationally easy. The idea of the algorithm is to sample an integer $p \in [\![0; n-1]\!]$ and a permutation $\sigma \in \mathfrak{S}_n$ independently and to start over if the p largest elements of the permutation are incorrectly sorted. The generation of the permutation can be done using the Fisher-Yates algorithm [FY48] which is asymptotically optimal. Moreover, the probability that a uniform such pair corresponds to a variation is $\frac{v_n}{n \cdot n!} \sim \frac{e}{n}$. Hence, in expectation, a linear number of rejections will occur before a valid variation is obtained. Since the Fisher-Yates algorithm is linear in terms of memory accesses and is $O(n \ln(n))$ is terms of random bits consumption, the overall procedure described here performs $O(n^2)$ memory accesses and consumes $O(n^2 \ln(n))$ random bits in expectation.

In its naive form, the algorithm we just described is thus less efficient than Algorithm 37. However, it can be improved significantly by using *early rejection*. Early rejection consists in aborting the generation as soon as we can decide that the object being generated will not result in a valid variation. In our case, this consists in aborting as soon as two integers larger than $n-p$ are encountered in the wrong order.

In fact, the algorithm can be made even faster by first sampling the relative ordering of these p elements, and then sampling the rest of the permutation. Since only one ordering is valid, this actually boils down to sampling a Bernoulli variable of parameter $\frac{1}{p!}$ and aborting the generation if it is equal to 0. If this test passes, the resulting variation is obtained by shuffling an array containing all the integers from 1 to $n - p$ and p zeros. This is described in Algorithm 38.

Algorithm 38 Uniform pre-computation-free random sampler of variations based on the rejection principle.

Input: An integer $n > 0$
Output: A uniform random variation of size n

```
function UnifVariationRej(n)
    p ⟵ Unif(⟦0; n − 1⟧)
    for q from p down to 2 do
        if not Bern(1/q) then return UnifVariationR(n)
    return UnifPermBounded(n, p)
```

Input: Two integers n and p such that $0 \leq p < n$
Output: A uniform permutation of size n whose p largest elements are replaced by zeros

```
function UnifPermBounded(n, p)
    T ⟵ [1, 2, 3, …, n − p, 0, 0, …, 0]
    for i from 0 to n − 2 do
        j ⟵ Unif(⟦i; n − 1⟧)
        T[i] ↔ T[j]                                  ▹ swap
```

Note that Algorithm 38 implements the Bernoulli variable of parameter $\frac{1}{p!}$ by sampling $(p-1)$ Bernoulli variables of parameters $\frac{1}{p}, \frac{1}{p-1}, \ldots, \frac{1}{2}$ in order to avoid the costly computation of the factorial. Thanks to the early rejection principle, Algorithm 38 draws less random variables than its naive counterpart (which samples full permutations before the validity check), and only performs $(n-1)$ swaps. In fact, this algorithm is asymptotically optimal (in expectation), as stated by Theorem 24.

Theorem 24 (Complexity of Algorithm 38)**.** *To sample a variation of size n, Algorithm 38 performs $(n - 1)$ swaps and consumes $n \log_2(n) + O(n)$ random bits in expectation.*

Since we have $\log_2(v_n) = n \log_2(n) + O(n)$, the number of random bits necessary to sample a uniform variation is asymptotically lower bounded by $n \log_2(n)$, hence the optimality of Algorithm 38. Similarly, at least n memory accesses are necessary to write a variation.

To give a rough idea of the performance of this algorithm, our implementation can generate permutations of size $n = 10^7$ in about 0.16 seconds on a standard personal laptop. This is significantly better than the approach based on the recursive method. A more precise benchmark is given in Figure 3.5. For n ranging from 0 to 10^7 by steps of 10^5 we computed the average runtime of Algorithm 38 over 100 samples.

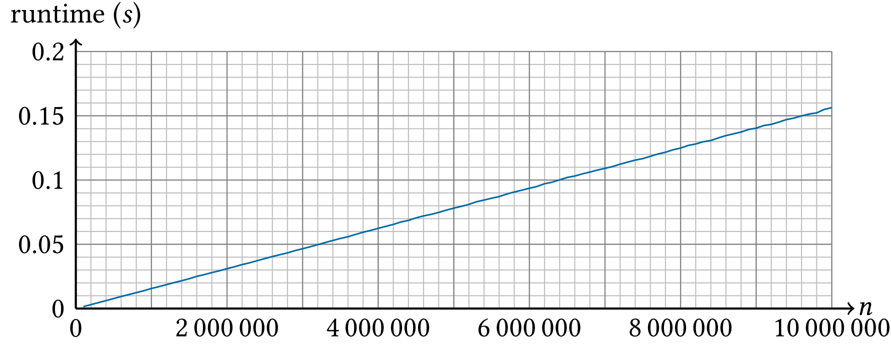


Figure 3.5: Runtime of the rejection-based uniform random sampler of variations (Algorithm 38).

Proof of Theorem 24. The shuffling part of the algorithm, which is implemented in function UnifPermBounded, consumes $n\log_2(n) + O(n)$ random bits in expectation. So we have to prove that the first part of the algorithm is linear.

The number of rejections of the first part follows a geometric distribution of parameter p_n where the probability of rejection p_n satisfies:

$$p_n = \frac{1}{n}\sum_{p=0}^{n-1}\frac{1}{p!} = \frac{e}{n} + O\left(\frac{1}{(n+1)!}\right).$$

Hence, the expected number of rejections is $\frac{1}{p_n} \sim \frac{n}{e}$. Furthermore, the cost in terms of random bits consumption of the for loop sampling several Bernoulli variables is bounded by

$$C + \frac{C}{p} + \frac{C}{p(p-1)} + \cdots + \frac{C}{p(p-1)(p-2)\cdots 3} = O(1)$$

where C is the (constant) cost of sampling the Bernoulli variables. □

3.3.3 Lexicographic unranking

We now turn to the problem of unranking variations lexicographically. We recall that a variation $v = (v_i)_{1\le i\le n}$ is said to be lexicographically smaller than another variation $v' = (v'_i)_{1\le i\le n}$ if there exists an index i_0 such that $v_i = v'_i$ for all $i < i_0$ and $v_{i_0} < v'_{i_0}$. We will only compare variations of the same size in this section. As an example, Table 3.2 on the next page lists all variations of size 3 in lexicographic order.

In order to unrank variations in lexicographic order, we need to know the number of variations starting by j for all $j \in [\![0; n]\!]$. In addition, once we know the first element j of a variation, the remaining $(n-1)$ elements can be reinterpreted as a variation with at least $(j-1)$ non-zero elements. Thus, we also need to know the number of (possibly empty) variations with at least p non-zero elements. Both quantities are given in Proposition 6 below.

Proposition 6. *Let n, p and j three integers such that $0 \le j, p \le n$ and $n > 0$. Let $v_{n,p}$ denote the number of (possibly empty if $p = 0$) variations with at least p non-zero elements and let $v^{(j)}_{n,p}$ denote*

Table 3.2: All variations of size 3 and their rank for the lexicographic order

rank	variation
0	(0, 0, 1)
1	(0, 1, 0)
2	(0, 1, 2)
3	(0, 2, 1)
4	(1, 0, 0)

rank	variation
5	(1, 0, 2)
6	(1, 2, 0)
7	(1, 2, 3)
8	(1, 3, 2)
9	(2, 0, 1)

rank	variation
10	(2, 1, 0)
11	(2, 1, 3)
12	(2, 3, 1)
13	(3, 1, 2)
14	(3, 2, 1)

the number of such variations starting by j. We have that

$$v_{n,p} = v_{n-p,0} \cdot n \cdot (n-1) \cdot (n-2) \cdots (n-p+1) \tag{3.6}$$

$$v_{n,0} = 1 + v_n \tag{3.7}$$

$$v_{n,p}^{(0)} = \mathbb{1}_{\{p<n\}} \cdot v_{n-1,p} \tag{3.8}$$

$$v_{n,p}^{(j)} = v_{n-1,\max(j-1,p-1)} \qquad \textit{when } j > 0 \tag{3.9}$$

Proof. A variation with at least p non-zero elements can be decomposed in two parts: its p largest elements on one hand, and the remaining of the variation on the other hand. The first part is counted by $n \cdot (n-1) \cdots (n-p+1)$ because the largest element can be anywhere in the array (n options), the second largest can be anywhere in the $(n-1)$ remaining positions, etc. The rest of the variation forms a possibly empty variation of size $(n-p)$. Hence the formula (3.6).

A possibly empty variation of size n is either an array of zeros (there is only one of them) or a non-empty variation, hence (3.7).

A variation may start by a zero only if it is not full, that is if $p < n$. Moreover, the $(n-1)$ remaining elements form a variation of size $(n-1)$ whose number of non-zero elements is also p. Hence (3.8).

Finally, the $(n-1)$ remaining elements of a variation starting by $j > 0$ can be seen as a variation of size $(n-1)$ by decrementing by 1 all of its elements that are larger than j. This variation has at least $(j-1)$ non-zero elements since all the integers from 1 to $(j-1)$ must be present. Moreover, if the overall variation of size n has at least p non-zero elements, then the sub-variation of size $(n-1)$ has at least $(p-1)$ of them. Hence (3.9). □

The idea for unranking a variation of size n, with at least p non-zero elements, and of rank u (with $0 \leq u < v_{n,p}$) is the following. First we find the first element of the variation, this is achieved by computing the smallest j such that $r < \sum_{i=0}^{j} v_{n,p}^{(i)}$. Then we unrank a variation of size $(n-1)$, with $\max(p-1, j-1)$ non-zero elements and of rank $r - \sum_{i=0}^{j-1} v_{n,p}^{(i)}$ and we increment by one the elements of this variation that are greater or equal to j. Finally, we insert j in the first position of this variation. We start with $p = 1$ to get a non-empty variation at the end of the procedure.

A straightforward implementation of this procedure, written in an imperative style, is presented in Algorithm 39. For the sake of clarity, we give here a naive though easy to understand algorithm. At the end of this section, we will introduce several optimisations to make this procedure significantly faster. Note that rather than inserting j at the current position and incre-

menting the latter occurrences of integers greater than j, as described above, we maintain a set of not-yet-inserted integers from which we pick the j-th element.

Algorithm 39 Lexicographic unranking of variations

Input: Two integers n and r such that $0 \leq r < v_n$
Output: The r-th variation of size n in lexicographic order

```
function UnrankVar(n, r)
    V ← [0, 0, ..., 0] (of length n),    S ← {1, 2, ..., n},    p ← 1
    for i from 1 to n do
        j ← 0
        C ← v_{n,p}^{(j)}
        while r ≥ C do
            r ← r − C
            j ← j + 1
            C ← v_{n,p}^{(j)}
        if j = 0 then V[i − 1] ← 0
        else
            V[i − 1] ← the j-th smallest element of S (counting from 1)
            remove V[i] from S
            p ← max(j − 1, p − 1)
    return V
```

This algorithm requires to compute many coefficients of the form $v_{n,p}^{(j)}$, which can be costly. One option is to pre-compute them all to have access to them in constant time. This is relevant if one needs to unrank many variations but this requires a significant amount of memory which can quickly become limiting.

We first establish the complexity of this algorithm in terms of number of coefficient evaluations. This is a valid measure of complexity as it also corresponds to the number of comparisons and to the number of arithmetic operations (subtractions) on r. At the end of this section, we describe an optimised implementation where no pre-computations are necessary and where the coefficients are computed on the fly at the cost of $O(n \cdot \mathcal{M}(\log_2 n))$ bitwise operations.

Theorem 25 (Complexity of Algorithm 39)**.** *The number of coefficients of the form $v_{n,p}^{(j)}$ evaluated by Algorithm 39 for unranking a variation of size n is $\frac{n^2+3n}{2}$ in the worst case and $\frac{n^2+5n-1}{4} + o(1)$ in average.*

Proof. The worst case corresponds to the last variations, that is $v = (n, n-1, n-2, \dots, 1)$. In that case, at iteration i of the for loop (starting from $i = 1$), the algorithm evaluates 1 coefficient before the while loop and $(n+1-i)$ coefficients in the while loop before finding that $v_i = n+1-i$. Hence, the cost of this execution is $\sum_{i=1}^{n}(n-i+2) = \sum_{i=2}^{n+1} i = \frac{n^2+3n}{2}$.

For studying the average complexity of the algorithm, we first compute the cumulated number of coefficients evaluated by the algorithm while unranking all variations of size n and with exactly p non-zero components. Let $u_{n,p}$ denote this number. For unranking such a variation $v = (v_i)_{1 \leq i \leq n}$, the algorithm first needs to determine the value of v_1. To achieve this, it

first evaluates one coefficient before the while loop and then it performs v_1 iterations of the while loop to find v_1. This accounts for $(1 + v_1)$ coefficient evaluations. Then, the rest of the execution corresponds to unranking the variations $v' = (v'_i)_{2\leq i\leq n}$ where $v'_i = v_i - 1$ if $v_i > v_1 > 0$ and $v'_i = v_i$ otherwise. Note that the two following statements hold:

- When v describes all the variations of size n, with p non-zero components, and such that $v_1 = 0$, then v' describes all variations of size $n - 1$ with p non-zero components.
- When v describes all the variations of size n, with p non-zero components, and such that $v_1 = j > 0$, then v' describes all variations of size $n - 1$ with $p - 1$ non-zero components.

Finally, note that the total number of variations of size n with p non-zero components is given by $p!\binom{n}{p}$, and the number of such variations starting with $v_1 = j > 0$ is $(p-1)!\binom{n-1}{p-1}$. As a consequence we have that for all $n > 0$

$$\begin{aligned} u_{n,p} &= p!\binom{n}{p} + u_{n-1,p} + \sum_{j=1}^{p}\left(j\cdot(p-1)!\binom{n-1}{p-1} + u_{n-1,p-1}\right) \\ &= p!\binom{n}{p} + \frac{(p+1)!}{2}\binom{n-1}{p-1} + u_{n-1,p} + p\cdot u_{n-1,p-1}. \end{aligned} \tag{3.10}$$

In order to simplify the end of the computations, we introduce the following generating functions:

$$\begin{aligned} U(x,y) &= \sum_{n>0}\sum_{p=0}^{n}\frac{u_{n,p}}{p!}x^n y^p \\ A(x,y) &= \sum_{n>0}\sum_{p=0}^{n}\left(\binom{n}{p} + \frac{p+1}{2}\binom{n-1}{p-1}\right)x^n y^p. \end{aligned}$$

From (3.10), we get:

$$U(x,y) = A(x,y) + xU(x,y) + xyU(x,y) = \frac{A(x,y)}{1-x(1+y)}.$$

Moreover

$$\begin{aligned} A(x,y) &= \frac{x(1+y)}{1-x(1+y)} + \sum_{n>0}\sum_{p=1}^{n}\binom{n-1}{p-1}x^n y^p + \sum_{n>0}\sum_{p=1}^{n}\frac{p-1}{2}\binom{n-1}{p-1}x^n y^p \\ &= \frac{x(1+y)}{1-x(1+y)} + \frac{xy}{1-x(1+y)} + \sum_{n\geq 0}\sum_{p=0}^{n}\frac{p}{2}\binom{n}{p}x^{n+1}y^{p+1} \\ &= \frac{x+2xy}{1-x(1+y)} + \sum_{n\geq 0}\sum_{p=0}^{n}\frac{n+1}{2}\binom{n}{p}x^{n+2}y^{p+2} \\ &= \frac{x+2xy}{1-x(1+y)} + \frac{x^2y^2}{2(1-x(1+y))^2} \end{aligned}$$

so that finally:

$$U(x,y) = \frac{x+2xy}{(1-x(1+y))^2} + \frac{x^2y^2}{2(1-x(1+y))^3}.$$

We recover $u_{n,p}$ by extracting the coefficient in front of $x^n y^p$ in the above equality. This yields:

$$\begin{aligned}\frac{u_{n,p}}{p!} &= n\binom{n-1}{p} + 2n\binom{n-1}{p-1} + \frac{n(n-1)}{4}\binom{n-2}{p-2} \\ &= \binom{n}{p}\left(n - p + 2p + \frac{p(p-1)}{4}\right).\end{aligned}$$

In order to get the average complexity of the algorithm, we need to sum the contributions of all the variations with p non-zero components for all $p > 0$, that is

$$\begin{aligned}\sum_{p=1}^{n} u_{n,p} = \sum_{p=0}^{n-1} u_{n,n-p} &= \sum_{p=0}^{n} \frac{n!}{p!}\left(2n - p + \frac{(n-p)(n-p-1)}{4}\right) \\ &= \sum_{p=0}^{n} \frac{n!}{p!}\left(\frac{n^2+7n}{4} - \frac{n+1}{2}p + \frac{p(p-1)}{4}\right) \\ &= \frac{n^2+7n}{4}v_n - \frac{n+1}{2}(v_n - n) + \frac{1}{4}(v_n - n^2) \\ &= \frac{n^2+5n-1}{4}v_n + \frac{n^2+2n}{4}.\end{aligned}$$

We can then conclude the proof by dividing by the total number of variations of size n, which yields an average complexity of

$$\frac{n^2+5n-1}{4} + \frac{n^2+2n}{4v_n} = \frac{n^2+5n-1}{4} + O\left(\frac{1}{(n-2)!}\right). \qquad \square$$

Faster unranking by reusing computations

In order to speed up the generation in a memory-limited context, or when n is too large for the pre-computations to be tractable, we need to compute the $v_{n,p}^{(j)}$ coefficients on the fly. If computed naively using our recurrence formulas, computing $v_{n,p}^{(j)}$ requires $O(n^2\mathcal{M}(\log_2 n))$ arithmetic operations.

The idea to reduce the cost of one evaluation is to re-use the result from the previous evaluation. This is possible due to the fact that the parameters n, p and j vary only by one (at most) from one evaluation to the following. The relations we give below on coefficients with close parameters allow to take advantage of this fact in the algorithm.

Proposition 7. *For all $n > 1$ and p such that $0 \le p < n$, we have:*

$$v_{n,p} = n \cdot v_{n-1,p} + \frac{n!}{(n-p)!} = v_{n,p+1} + \frac{n!}{(n-p)!}$$

Proof. Let $n > 1$ and $0 \le p < n$. The variations with at least p non-zero elements either have exactly p non-zero elements or have at least $(p+1)$ of them. The former is counted by $n \cdot (n-1) \cdots (n-p+1)$[4] and the latter is counted by $v_{n,p+1} = n \cdot v_{n,p}$. $\square$

[4]This is also called the number of p-arrangements of n.

Using the relations from Proposition 7, we can compute the values of $v_{n,p}^{(j)}$ quickly in Algorithm 39 by re-using the value of C from one evaluation to the next and by keeping a product of the form $\frac{n!}{(n-p)!} = n \cdot (n-1) \cdots (n-p+1)$ in an auxiliary variable P. Each time p, j or i are modified, the variables C and P are updated at the cost of a constant number of multiplications or divisions by a small integer. This optimisation is presented in Algorithm 40.

Algorithm 40 Optimised unranking procedure of variations

```
Input: Two integers n and r such that 0 ≤ r < v_n
Output: The r-th variation of size n in lexicographic order
  function UnrankVar(n, r)
    V ⟵ [0, 0, ..., 0], S ⟵ {1, 2, ..., n}                ▷ both of length n
    p ⟵ 1, C ⟵ v_{n−1}, P ⟵ 1
    for i from 0 to n − 2 do     ▷ Invariants: (1) C = v_{n−1−i,p}   (2) P = ∏_{k=n+1−p−i}^{n−1−i} k
      if r < C then
        V[i] ⟵ 0
        if p > 0 then P ⟵ P · (n − i − p)
      else
        r ⟵ r − C, j ⟵ 1
        if p > 0 then C ⟵ C + P
        while r ≥ C do                   ▷ Invariant: P = ∏_{k=n−i−max(j,p)+1}^{n−i−1}
          r ⟵ r − C
          if j ≥ p then
            C ⟵ C − P
            P ⟵ P · (n − j − i)
          j ⟵ j + 1
        V[i] ⟵ the j-th smallest element of S (counting from 1)
        remove V[i] from S
        p ⟵ max(p, j) − 1
      C ⟵ (C − P)/(n − i − 1)            ▷ At this point we have P = ∏_{k=n−p−i}^{n−i−1} k
      if p > 0 then P ⟵ P/(n − i − 1)
    if p = 1 ∨ r = 1 then V[n − 1] ⟵ the smallest element of S
    else V[n − 1] ⟵ 0
    return V
```

As we saw before, the cost of the multiplication of an integer of bit-size bounded by $\log_2(n)$ with an integer of size $O(n \log_2(n))$ is in $O(n\mathcal{M}(\log_2(n)))$ by Lemma 1 on page 6. Hence, since in the worst case we have $O(n^2)$ such operations (according to Theorem 25), the bit-complexity of Algorithm 40 is $O(n^3\mathcal{M}(\log_2(n)))$.

As a final remark, note that in practice the integers of bit-size bounded by $\log_2(n)$ will fit in a machine word so that the cost of one multiplication is in fact of the form $O(n \log_2(n))$ in terms of arithmetic operations on *machine words*. As a consequence, the runtime of algorithm can be expected to behave like $n^3 \log_2(n)$ in practice.

3.4 Usainboltz: fast and generic Boltzmann sampling

In this section we study the implementation of Boltzmann samplers from a practical perspective. The Boltzmann sampling framework is usually presented from a high level perspective as a *compiler* which takes a specification as an input and, following simple rules, produces an algorithm. Although this point of view works well for writing a sampler for a particular class, a more "dynamic" approach is preferable for the purpose of writing a generic Boltzmann sampling library.

The first contribution of this section is to describe in detail how to implement a generic general-purpose Boltzmann sampler which takes a specification as an input and outputs objects directly without needing a compilation phase. This approach is not new and is already used in practice, e.g. in [2] and [5]. However the low level aspects of its implementation have never been described in detail and Section 3.4.1 is meant as a guide on how to implement such an algorithm. Moreover, we also present here some implementation details and new optimisations which provide a significant speed-up in practice. A second aspect of writing a Boltzmann sampling library is to make it easy to integrate into an other code-base. To this end, we introduce the notion of builders which takes advantage of the recursive nature of Boltzmann samplers and borrows ideas from functional programming [MFP91] to give the end-user a convenient way to control how the generated objects are built. This is presented in Section 3.4.2.

The ideas presented here have led to the implementation of a Python Boltzmann sampling library *usainboltz: Uniform SAmplINg with BOLTZmann* [6] by Matthieu Dien and the author of this thesis. It is based on a fast C++ core and offers a generic interface allowing to integrate it, in particular, into the Sagemath [4] computer algebra system. Some ideas were also borrowed to [5] which served as a playground for testing alternative implementations and experimenting ideas.

3.4.1 Practical implementation

We start by formalising the objects that we will use as an input. The base objects on which we will work in this section are systems of recursive combinatorial specifications of the form

$$\begin{cases} \mathcal{F}_1 = \Psi_1(\mathcal{Z}, \mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_k) \\ \mathcal{F}_2 = \Psi_2(\mathcal{Z}, \mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_k) \\ \dots \\ \mathcal{F}_k = \Psi_k(\mathcal{Z}, \mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_i) \end{cases} \tag{3.11}$$

where the $\Psi_i(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k)$ are any functional terms involving only the usual constructors of analytic combinatorics $\{+, \times, \mathcal{E}, \textsc{Seq}, \dots\}$ and the classes $\mathcal{E}$, $\mathcal{Z}$ and $\mathcal{F}_i$. In order to prepare the ground for the introduction of builders later in Section 3.4.2, we make this a little more formal. We first define inductively a set $\mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)$ of terms built on the list of symbols $Z, Z_1, Z_2, \dots, Z_k$ and E. They are defined in Figure 3.6 on the next page. Those are purely syntactic objects. We then see the specifications appearing in (3.11) as an interpretation of these syntactic expressions (see below).

$$\frac{}{Z \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)} \qquad \frac{}{E \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)} \qquad \frac{}{Z_i \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)}$$

$$\frac{\Psi, \Psi' \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)}{\Psi + \Psi' \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)} \qquad \frac{\Psi, \Psi' \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)}{\Psi \times \Psi' \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)}$$

$$\frac{\Psi \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k) \qquad \text{Constr} \in \{\text{Seq}, \text{MSet}\}}{\text{Constr}(\Psi) \in \mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)}$$

Figure 3.6: Inference rules defining the set of syntactic expressions $\mathfrak{F}(Z, Z_1, Z_2, \dots, Z_k)$.

$$\begin{aligned}
E(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) &= \mathcal{E} \\
Z(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) &= \mathcal{Z} \\
Z_i(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) &= \mathcal{F}_i \\
\text{Seq}(\Psi)(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) &= \text{Seq}(\Psi(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k)) \\
\text{MSet}(\Psi)(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) &= \text{MSet}(\Psi(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k)) \\
(\Psi + \Psi')(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) &= \Psi(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) + \Psi'(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) \\
(\Psi \times \Psi')(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) &= \Psi(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k) \times \Psi'(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k)
\end{aligned}$$

Figure 3.7: Interpretation of an expression $\Psi \in \mathfrak{F}(Z, Z_1, \dots, Z_k)$ as a combinatorial class.

Given k combinatorial classes $\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_k$ and an expression $\Psi \in \mathfrak{F}(Z, Z_1, \dots, Z_k)$, we define the combinatorial class $\Psi(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k)$ inductively by interpreting each of the Z and Z_i symbols as $\mathcal{Z}$ and $\mathcal{F}_i$ and the constructions $+$, $\times$, ... appearing in Ψ, as the disjoint union, the Cartesian product of analytic combinatorics, etc. This is described in Figure 3.7. This gives a formal definition to the combinatorial system (3.11) and the syntactic expressions Ψ_i give a convenient base for defining builders in Section 3.4.2. Our goal, from now on is to describe an algorithm taking such a specification as an input and generating elements of $\mathcal{F}_1$ in the Boltzmann model.

We have already mentioned early rejection, which consists in aborting the generation as soon as we know that its result will be "too big", as a way to remedy the default of performance of rejection samplers in the case of *peaked* distributions. In fact, implementing it always has a positive impact on performance, whatever the type of objects. Here we present a second optimisation, which does not appear in the original article [DFLS04] and does not seem to be present in existing implementations such as [2].

Simulation

As we mentioned earlier, in most cases of interest, rejection samplers will reject several structures before finding one with a size in the target interval $I(n, \epsilon)$. In [DFLS04], the complexity measure

used to establish the linearity of these samplers is the number of arithmetic operations on real numbers. However, in practice and especially in the context of garbage-collected languages, most of the runtime is spent in the construction of the objects in memory, whereas arithmetic computations remain cheap. While the number of memory allocations and accesses is linear too, it can be reduced from $O(n)$ to exactly n by first "simulating" the generation, that is only computing the *size* N of the objects being generated rather than actually constructing them in memory. Once a size $N \in I(n, \epsilon)$ has been obtained, the pseudo-random number generator[5] (PRNG) used for the generation must be reset to its state just before the simulation and actual Boltzmann generating can take place using the free version of the sampler.

Similarly to free Boltzmann samplers, a simulation algorithm can be derived automatically from the specification of a class using simple rules. This is described in Table 3.3.

Table 3.3: Classical unlabelled operators of the symbolic method and their Boltzmann simulators Sim$[\mathcal{C}]$

Class	Spec.	Boltzmann simulator Sim$[\mathcal{C}](z)$
Neutral	$\mathcal{E}$	0
Atom	$\mathcal{Z}$	1
Union	$\mathcal{A} + \mathcal{B}$	**if** $\text{Bern}\left(\frac{A(z)}{A(z)+B(z)}\right)$ **then** $\text{Sim}[\mathcal{A}](z)$ **else** $\text{Sim}[\mathcal{B}](z)$
Product	$\mathcal{A} \times \mathcal{B}$	$\text{Sim}[\mathcal{A}](z) + \text{Sim}[\mathcal{B}](z)$
Sequence	$\textsc{Seq}(\mathcal{A})$	$k \longleftarrow \text{Geom}(1 - A(z)); \sum_{i=1}^{k} \text{Sim}[\mathcal{A}](z)$
Multi-set	$\textsc{MSet}(\mathcal{A})$	cf. Algorithm 41

Algorithm 41 Simulator for the multi-set construction $\textsc{MSet}(\mathcal{A})$

function $\text{Sim}[\textsc{MSet}(\mathcal{A})](z)$
  $M \longleftarrow 0$
  $k_0 \longleftarrow$ drawn according to $\mathbb{P}[k] = \exp\left(\frac{1}{k}A(z^k)\right) / \exp\left(\sum_{j>0} \frac{1}{j}A(z^j)\right)$
  for j **from** 1 **to** $k_0 - 1$ **do**
    for i **from** 1 **to** $\textsc{Pois}\left(\frac{A(x^j)}{j}\right)$ **do** $M \longleftarrow M + j \cdot \text{Sim}[\mathcal{A}](z^j)$
  for i **from** 1 **to** $\textsc{Pois}_{\geq 1}\left(\frac{A(x^{k_0})}{k_0}\right)$ **do** $M \longleftarrow M + k_0 \cdot \text{Sim}[\mathcal{A}](z^{k_0})$
  return M

For instance, for the case of NFJ programs specified by (3.3), the simulation algorithm derived using the rules from Table 3.3 is given in Algorithm 42.

Note that some extra arithmetic operations on real numbers are incurred by this process since the final generation happens twice: first in the form of a simulation, and then by actually

[5] In the case where a "true" source of randomness is used in place of a PRNG, resetting the source of randomness to its previous state is not possible. In that case, as an alternative strategy, the outcome of the random variable generators (Bern, Geom, ...) dictating the generation can be stored in memory so that they can be replayed later. This approach has the important drawback that numerous memory accesses are still being performed during the simulation, however this is still more memory efficient than explicitly constructing trees structures.

Algorithm 42 Simulation algorithm for NFJ programs

Input: A number $z \in [0; 1/12]$.
Output: The size of an NFJ program generated from the Boltzmann model of parameter z
function $\mathrm{Sim}[\mathcal{F}](z)$
 if $\textsc{Bern}\left(\frac{6z}{1-\sqrt{1-12z}}\right)$ **then return** 1
 else return $\mathrm{Sim}[\mathcal{F}](z) + \mathrm{Sim}[\mathcal{F}](z)$ ▹ The three other cases are similar

constructing the object. In practice, this trade-off is beneficial as memory allocations are several order of magnitude more expensive than integer arithmetic.

A generic stack-machine

As mentioned earlier, the usual way to present the framework of Boltzmann sampling is to present it in a *meta-programming* style, that is as a way to generate an *algorithm* from a grammar. This algorithm can then be run to generate objects described by the grammar. This is also the approach taken by [2], using the `compile` sub-command.

For the purpose of implementing Boltzmann samplers as a generic library, we propose here a slightly different presentation as a single algorithm which dynamically decides which rule from Table 3.1 on page 93 to apply by recursively traversing the specification. Moreover, in order to avoid any stack-explosion issue due to recursion, we propose an iterative implementation, which we present in the form of two stack-machines, one for the simulation and one for the generation.

The simulation stack-machine operates on pairs of the form of $\langle S, N\rangle_j^m$ where:

- N is an integer counting the number of already-generated atoms;
- m is an upper bound on the size: if N becomes larger than m, the simulation is aborted;
- j is an integer corresponding to the current multiplicity of the atoms;
- S is a stack of expressions or integers, that is:

$$
\begin{array}{rcll}
S & ::= & \varnothing & \\
 & | & \Psi : S & \text{where } \Psi \in \mathfrak{F}(Z, Z_1, \dots, Z_k) \\
 & | & j : S & \text{where } j \in \mathbb{N}^\star
\end{array}
$$

 where $\varnothing$ represents the empty stack, $x : S$ represents the stack whose first element is x and whose remaining elements are in S, and the Z_i are "non-terminal" symbols representing the classes $\mathcal{F}_i$.

The behaviour of the simulation machine is fully specified by the rewriting rules given in Table 3.4 on the following page. The simulation of the generation of an element of $\mathcal{F}_i$ starts at the term $\langle Z_i, 0\rangle_1^m$ and consists in applying the rewriting rules until an integer is obtained. The resulting integer is the size of the objects whose generation has been simulated and must be discarded if it exceeds m.

Note that the simulation needs to evaluate some expressions of the form $\Psi(\omega)$, which we define here. Given k combinatorial classes $\mathcal{F}_i$ and given a number z, the evaluation $\Psi(\omega)$ of an expression $\Psi \in \mathfrak{F}(Z, Z_1, \dots, Z_k)$ at ω and for these classes is defined as the evaluation of the generating function of $\Psi(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k)$ at ω. From a practical perspective, this requires that the values of the generating functions $F_i(z)$ of the $\mathcal{F}_i$ at z (and possibly at z^j for all $j > 0$) are known.

Table 3.4: Rewriting rules for the simulation stack-machine

Finish	$\langle \emptyset, N\rangle_j^m \longrightarrow N$
Multiplicity	$\langle j{:}S, N\rangle_i^m \longrightarrow \langle S, N\rangle_j^m$
Neutral	$\langle E{:}S, N\rangle_j^m \longrightarrow \langle S, N\rangle_j^m$
Atom	$\langle Z{:}S, N\rangle_j^m \longrightarrow$ **if** $N + j > m$ **then** $N + j$ **else** $\langle S, N + j\rangle_j^m$
Recursion	$\langle Z_i{:}S, N\rangle_j^m \longrightarrow \langle \Psi_i{:}S, N\rangle_j^m$
Union	$\langle (\Psi + \Psi'){:}S, N\rangle_j^m \longrightarrow$ **if** $\textsc{Bern}\left(\frac{\Psi(z^j)}{\Psi(z^j)+\Psi'(z^j)}\right)$ **then** $\langle \Psi{:}S, N\rangle_j^m$ **else** $\langle \Psi'{:}S, N\rangle_j^m$
Product	$\langle (\Psi \times \Psi'){:}S, N\rangle_j^m \longrightarrow \langle \Psi{:}\Psi'{:}S, N\rangle_j^m$
Sequence	$\langle \textsc{Seq}(\Psi){:}S, N\rangle_j^m \longrightarrow$ **let** $k = \textsc{Geom}(1 - \Psi(z^j))$ **in** $\langle \Psi{:}\Psi{:}\cdots{:}\Psi{:}S, N\rangle_j^m$ (k times)
Multi-set	cf. Figure 3.8

$$
\begin{aligned}
\langle \textsc{MSet}(\Psi){:}S, N\rangle_\ell^m \longrightarrow\quad & \text{draw } k_0 \text{ according to } \mathbb{P}[k] = \exp\left(\tfrac{1}{k}A(z^k)\right)/\exp\left(\textstyle\sum_{j>0}\tfrac{1}{j}A(z^j)\right)\\
& S' \longleftarrow \ell{:}S\\
& \textbf{for } j \textbf{ from } 1 \textbf{ to } k_0 - 1\\
& \quad S' \longleftarrow (j\cdot\ell){:}\Psi{:}\Psi{:}\cdots{:}\Psi{:}S' \quad \left(\textsc{Pois}\left(\tfrac{A(x^j)}{j}\right) \text{ times}\right)\\
& S' \longleftarrow (k_0\cdot\ell){:}\Psi{:}\Psi{:}\cdots{:}\Psi{:}S' \quad \left(\textsc{Pois}_{\geq 1}\left(\tfrac{A(x^{k_0})}{k_0}\right) \text{ times}\right)\\
& \langle S', N\rangle_\ell^m
\end{aligned}
$$

Figure 3.8: Rewriting rule for the simulation of $\textsc{MSet}(\Psi)$

The generation stack-machine has a similar behaviour but it must also actually build trees. It differs from the simulation machine on two aspects. First, rather than operating on an integer representing a size, it operates on a stack of already-built sub-trees. Theses trees correspond to sub-structure that would be produced by the recursive calls of the naive sampler before they are recombined as final objects. And second, we need to extend the set of symbols used in the stack S with four special symbols: a symbol `PROD` indicating that a pair must be formed with the two last generated trees, a symbol `LIST`(k) indicating that a list must be formed with the k last generated trees, a symbol `SET`(k) indicating that a set must be formed with the k last generated trees, and a symbol `DUP`(k) indicating that the last generated tree must be duplicated k times. The machine is described in Table 3.5 on the following page as a system of rewriting rules on pairs of the form $\langle S, T\rangle_j$ where:

- S is a stack of special symbols or expression from $\mathfrak{F}(Z, Z_1, \ldots, Z_k)$;
- T is a stack of nested pairs, lists, and sets of two special values z and ϵ, encoding trees;
- j is a positive integer corresponding to the current multiplicity of the atoms.

The free generation of an element of $\mathcal{F}_i$ is achieved by starting from $\langle Z_i{:}\emptyset, \emptyset\rangle_1$ and applying the rewriting rules from Table 3.5 on the next page until a tree is obtained.

The two stack-machines we presented above can be combined to write an algorithm which takes a combinatorial specification, a target size n, and a tolerance ϵ as an input and returns an object as described by the specification of size in $I(n, \epsilon)$. This is presented in Algorithm 43. The tuning part of the algorithm, that is the pre-processing step which consists in finding a good value for the parameter z and computing the values of the different generating functions at these

Table 3.5: Rewriting rules for the generation stack-machine

Finish	$\langle \emptyset, t{:}\emptyset \rangle_j \longrightarrow t$
Multiplicity	$\langle j{:}S, N \rangle_i \longrightarrow \langle S, N \rangle_j^m$
Build pair	$\langle \mathtt{PROD}{:}S, t_1{:}t_2{:}T \rangle_j \longrightarrow \langle S, (t_1, t_2){:}T \rangle_j$
Build list	$\langle \mathtt{LIST}(k){:}S, t_1{:}t_2{:}\cdots{:}t_k{:}T \rangle_j \longrightarrow \langle S, [t_1, t_2, \ldots, t_k]{:}T \rangle_j$
Build set	$\langle \mathtt{SET}(k){:}S, t_1{:}t_2{:}\cdots{:}t_k{:}T \rangle_j \longrightarrow \langle S, \{t_1, t_2, \ldots, t_k\}{:}T \rangle_j$
Neutral	$\langle E{:}S, T \rangle_j \longrightarrow \langle S, \epsilon{:}T \rangle_j$
Atom	$\langle Z{:}S, T \rangle_j \longrightarrow \langle S, z{:}T \rangle_j$
Recursion	$\langle Z_i{:}S, T \rangle_j \longrightarrow \langle \Psi_i{:}S, T \rangle_j$
Union	$\langle (\Psi + \Psi'){:}S, T \rangle_j \longrightarrow$ **if** $\textsc{Bern}\left(\frac{\Psi(z^j)}{\Psi(z^j)+\Psi'(z^j)}\right)$ **then** $\langle \Psi{:}S, T \rangle_j$ **else** $\langle \Psi'{:}S, T \rangle_j$
Product	$\langle (\Psi \times \Psi'){:}S, T \rangle_j \longrightarrow \langle \Psi{:}\Psi'{:}\mathtt{PROD}{:}S, T \rangle_j$
Sequence	$\langle \textsc{Seq}(\Psi){:}S, T \rangle_j \longrightarrow$ **let** $k = \textsc{Geom}(1 - \Psi(z^j))$ **in** $\langle \Psi{:}\Psi{:}\cdots{:}\Psi{:}\mathtt{LIST}(k){:}S, T \rangle_j \quad (k \text{ times})$
Multi-set	cf. Figure 3.9

$\langle \textsc{MSet}(\Psi){:}S, T \rangle_\ell \longrightarrow$ draw k_0 according to $\mathbb{P}[k] = \exp\left(\frac{1}{k} A(z^k)\right) / \exp\left(\sum_{j>0} \frac{1}{j} A(z^j)\right)$

draw p_{k_0} according to $\textsc{Pois}_{\geq 1}\left(\frac{A(x^{k_0})}{k_0}\right)$

for j **from** 1 **to** $k_0 - 1$ draw p_j according to $\textsc{Pois}\left(\frac{A(x^j)}{j}\right)$

$S' \longleftarrow k{:}\mathtt{SET}(\sum j \cdot p_j){:}S$

for j **from** 1 **to** k_0

$S' \longleftarrow (j \cdot \ell){:}\Psi{:}\mathtt{DUP}(j){:}\Psi{:}\mathtt{DUP}(j){:}\ldots{:}\Psi{:}\mathtt{DUP}(j){:}S' \quad (p_j \text{ times})$

$\langle S', T \rangle_\ell$

Figure 3.9: Rewriting rule for the generation of $\textsc{MSet}(\Psi)$

points, is delegated to an external tool such as [3] or [11] and is omitted here.

3.4.2 Generic Boltzmann sampling with builders

We now turn to another practical aspect of Boltzmann sampling which is to make it easy to integrate within an existing code-base. In the algorithms we have presented so far, as well as in the current implementations of Boltzmann samplers, the generated objects have a uniform tree-like imposed representation. This means that users willing to use their own data-structure with an existing Boltzmann sampling library has to first generate an object and then traverse it again to convert it to the desired type.

In this section, we present a mechanism which allows the user to specify data constructors, in the form of functions, to be used during the generation. As a consequence, objects of the desired type are generated directly, without using an intermediate data-structure. We call these user-specified constructors "builders". The mechanism presented here borrows ideas from functional programming with recursion schemes [MFP91], to describe recursive functions from simple combinators.

Algorithm 43 Fast rejection sampler using the stack-machines from Table 3.4 on page 122 and Table 3.5 on the preceding page

Input: A specification $\mathcal{S}$ in the form of (3.11), a target size n and a tolerance ϵ
Output: A random element described by $\mathcal{S}$ following the Boltzmann distribution conditioned to have its size in $I(n, \epsilon)$

function GenInInterval($\mathcal{S}, n, \epsilon$)
  $s \longleftarrow$ a copy of the PRNG state
  $N \longleftarrow$ execute the simulation machine from the state $\langle Z_1 : \varnothing, 0\rangle_1^{(1+\epsilon)n}$
  while $|N/n - 1| > \epsilon$ **do**
    $s \longleftarrow$ a copy of the PRNG state
    $N \longleftarrow$ execute the simulation machine from the state $\langle Z_1 : \varnothing, 0\rangle_1^{(1+\epsilon)n}$
  restore the state of the PRNG to s
  $t \longleftarrow$ execute the generation machine from the state $\langle Z_1 : \varnothing, \varnothing\rangle_1$
  return t

Let $\Psi \in \mathfrak{F}(Z, Z_1, \dots, Z_k)$ be an expression. The combinatorial class $\Psi(\mathcal{Z}, \mathcal{F}_1, \dots, \mathcal{F}_k)$ obtained by substitution can be seen as the composition of the combinatorial class $\Psi(\mathcal{Z}, \mathcal{Z}_1, \dots, \mathcal{Z}_k)$ (where the $\mathcal{Z}_i$ are k distinct atoms) with $\mathcal{F}_1 \times \mathcal{F}_2 \times \cdots \times \mathcal{F}_k$. A property of Boltzmann samplers is that sampling an element from $\Psi(\mathcal{Z}, \mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_k)$ is equivalent to (1) sampling an element ψ from $\Psi(\mathcal{Z}, \mathcal{Z}_1, \dots, \mathcal{Z}_k)$ with the appropriate probability distribution and (2) for each i and for each occurrence of $z_i \in \mathcal{Z}_i$ in ψ, sampling an independent element from $\mathcal{F}_i$. The main idea of the builder mechanism is to take advantage of this property to apply a transformation after *each* recursive call generating an element of $\mathcal{F}_i$.

More precisely, let $X_1, X_2, \dots, X_k$ be any k sets, consider given a combinatorial system of the form (3.11), and say that every time we generate an object described by $\mathcal{F}_i$ we want it to be an element of X_i. One way to achieve this is to define k functions $f_i : \Psi(\mathcal{Z}, X_1, X_2, \dots, X_k) \longrightarrow X_i$ and to apply f_i to the result of each call to the $\mathcal{F}_i$ sampler (which thus yields elements of X_i). Concretely, an execution of the $\mathcal{F}_i$ sampler becomes:

1. generate an element ψ of $\Psi_i(\mathcal{Z}, \mathcal{Z}_1, \dots, \mathcal{Z}_k)$;
2. substitute each z_j in ψ with an independent element of X_j (and not $\mathcal{F}_i$) generated recursively using the $\mathcal{F}_j$ sampler (ψ now belongs to $\Psi(\mathcal{Z}, X_1, X_2, \dots, X_k)$);
3. apply f_i to ψ.

Note that the builders f_i implement a conversion from $\mathcal{F}_i$ to X_i without explicitly traversing the structure. They only perform one step of the conversion and the handling of the recursion is delegated to the generation algorithm. The generation algorithm thus implements a "fold" whose argument functions are the builders. Using the vocabulary from [MFP91], this corresponds to a *catamorphism* on the data-type $\mathcal{F}_i$. An important aspect of the builder mechanism is that the builders are applied on the fly during the generation.

We first illustrate the use of builders on a few examples and then we present how to modify the stack-machine introduced before to implement them.

Builders in action

Generating syntactic NFJ programs Consider again the example of the set of NFJ programs from Chapter 1 and recall that their specification is $\mathcal{F} = \mathcal{Z} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F}$. Say, we use the following system of expressions to describe $\mathcal{F}$:

$$\Psi_1 = Z + Z_2 + Z_3 + Z_4 \qquad \Psi_2 = Z_1 \times Z_1 \quad \text{(parallel composition)}$$
$$\Psi_3 = Z_1 \times Z_1 \quad \text{(sequential composition)} \qquad \Psi_4 = Z_1 \times Z_1 \quad \text{(non-deterministic choice)}$$

Assume we have four data constructors `Atom`, `Par`, `Seq` and `Choice` for the four constructions of the NFJ language and denote by X the set of the programs written using these constructors. Note that $\mathcal{F}$ denotes the *combinatorial class* modelling NFJ programs and that X is a set of *syntactic programs*. Here we want to generate elements of X. In order to achieve this we can let $X_1 = X_2 = X_3 = X_4 = X$ and use the following builders

$$f_1 = \begin{cases} \mathcal{Z} + X + X + X \longrightarrow X \\ x \longmapsto \texttt{Atom} \text{ if } x = z \text{ and } x \text{ otherwise} \end{cases} \qquad f_2 = \begin{cases} X \times X \longrightarrow X \\ x = (y, z) \longmapsto \texttt{Par}(y, z) \end{cases}$$
$$f_3 = \begin{cases} X \times X \longrightarrow X \\ x = (y, z) \longmapsto \texttt{Seq}(y, z) \end{cases} \qquad f_4 = \begin{cases} X \times X \longrightarrow X \\ x = (y, z) \longmapsto \texttt{Choice}(y, z) \end{cases}$$

Computing statistics without constructing the objects Another possible application of builders is to compute some statistics on the objects directly, without actually constructing them. For instance, in the case of NFJ programs, one can count the number of global choices of a program on the fly without generating it. This can be used to study the average number of global choices of NFJ programs experimentally without losing time in allocations. This can be achieved using the following builders and using $X_1 = X_2 = X_3 = X_4 = \mathbb{N}$.

$$f_1 = \begin{cases} \mathcal{Z} + \mathbb{N} + \mathbb{N} + \mathbb{N} \longrightarrow \mathbb{N} \\ n \longmapsto 1 \text{ if } x = z \text{ and } x \text{ otherwise} \end{cases} \quad f_2 = f_3 = \begin{cases} \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \\ (n, n') \longmapsto n \cdot n' \end{cases} \quad f_4 = \begin{cases} \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \\ (n, n') \longmapsto n + n' \end{cases}$$

On this example, we can see that no trees are ever constructed and that only integers are manipulated.

Applying bijections on the fly We end this series of examples with another possible application of builders. It is common in combinatorics that several combinatorial classes are counted by the same sequence. One of the most famous such sequences is the sequence of Catalan numbers referenced under `A000108` in the OEIS. One way of proving that two classes are counted by the same sequence is to establish a bijection preserving the size between them. Here we give two examples of families of objects, namely Dyck words and Plane Forests, which are counted by the Catalan sequence and can be generated using the *same* specification simply by using builders to apply the bijection on the fly.

The set $\mathcal{D}$ of Dyck words is defined as the set of well-parenthesised words on the alphabet $\{\texttt{(},\texttt{)}\}$. More formally, this is the set of words $w \in \{\texttt{(},\texttt{)}\}^\star$ containing as many opening and closing parentheses and such that all the prefixes of w contain at least as many `(` as `)`. A Dyck

word $w \in \mathcal{D}$ is either empty or can be uniquely decomposed as $w = (w')\,w''$ where w' and w'' are Dyck words too. As a consequence, taking the number of opening parentheses as the size of a word, we have the following specification $\mathcal{F}_1 = \mathcal{E} + \mathcal{Z} \times \mathcal{F}_1 \times \mathcal{F}_1$, which is also the specification of binary trees. A Boltzmann sampler generating Dyck words directly rather than tree-like structures is thus obtained by using the expression $\Psi_1 = E + Z \times Z_1 \times Z_1$ as a specification and the following builder:

$$f_1 = \begin{cases} \mathcal{E} + \mathcal{Z} \times \mathcal{D} \times \mathcal{D} \longrightarrow \mathcal{D} \\ x \mapsto \begin{cases} \text{the empty word} & \text{if } x = \epsilon \\ (w')\,w'' & \text{if } x = (z, w', w'') \end{cases} \end{cases}$$

Again, note that f_1 takes already built Dyck words as an input and only applies the final step of the construction of the word.

Now, more interestingly, the class $\mathcal{S}$ of plane forests is usually defined using a different specification but is in bijection with the class of Dyck words. A plane forest is defined as a possibly empty list of plane trees and a plane tree is defined as a node with a possibly empty list of children, or equivalently, a *forest* of children. Hence, taking the number of nodes of the forest as its size, the classes $\mathcal{S}$ and $\mathcal{T}$ of forests and trees are specified by:

$$\begin{aligned} \mathcal{S} &= \textsc{Seq}(\mathcal{T}) \\ \mathcal{T} &= \mathcal{Z} \times \mathcal{S} \end{aligned}$$

In order to see the bijection with the class of Dyck words, we need to present an alternative decomposition of plane forests. A forest $s = [t_1, t_2, \dots, t_k] \in \mathcal{S}$ is either empty ($k = 0$) or can be seen as a pair made of its first tree t_1 and the list of its remaining trees $[t_2, \dots, t_k]$. Moreover, the first tree t_1 can itself be seen as a forest (with a node on top). This yields a natural bijection ϕ between Dyck words and Plane Forests:

$$\phi = \begin{cases} \mathcal{D} \longrightarrow \mathcal{S} \\ w \mapsto \begin{cases} \text{the empty forest} & \text{if } w \text{ is the empty word} \\ [[t'_1, \dots, t'_\ell], t''_1, \dots, t''_k] & \text{if } w = (w')\,w'' \text{ where } \begin{matrix} [t'_1, \dots, t'_\ell] = \phi(w') \\ [t''_1, \dots, t''_k] = \phi(w'') \end{matrix} \end{cases} \end{cases}$$

Note that this bijection is defined recursively based on the decomposition of Dyck words presented above. It is thus possible to express it in the builder mechanism and to generate plane forest directly from the Dyck words generator. To this end, take $X_1 = \mathcal{S}$ and use the following builder instead of the Dyck words builder.

$$f_1 = \begin{cases} \mathcal{E} + \mathcal{Z} \times \mathcal{S} \times \mathcal{S} \longrightarrow \mathcal{S} \\ x \mapsto \begin{cases} \text{the empty forest} & \text{if } x = \epsilon \\ [[t'_1, \dots, t'_\ell], t''_1, \dots, t''_k] & \text{if } x = (z, [t'_1, \dots, t'_\ell], [t''_1, \dots, t''_k]) \end{cases} \end{cases}$$

Implementation of the builder mechanism

In order to implement the builder mechanism in the stack-machine from Table 3.5 on page 123, we need to add a new instruction $\texttt{BUILD}(i)$ which calls the i-th builder on the last generated object. Moreover, the rule that handles the Z_i symbols must insert $\texttt{BUILD}(i)$ in the stack of instructions so that it appears on top of the stack right after the object has been generated. The new rule and the updated rule for Z_i are given in Table 3.6.

Table 3.6: Rewriting rules implementing the builder mechanism in the sampling stack-machine

Build	$\langle \texttt{BUILD}(i) : S, t : T \rangle_j \longrightarrow \langle S, f_i(t) : T \rangle_j$
Recursion	$\langle Z_i : S, T \rangle_j \longrightarrow \langle \Psi_i : \texttt{BUILD}(i) : S, T \rangle_j$
Tag	$\langle \texttt{TAG}(i) : S, t : T \rangle_j \longrightarrow \langle S, t^{(i)} : T \rangle_j$

In addition to adding a builder instruction, it is also convenient for the user, when they write their builders, to have an indication of which alternative was chosen at each disjoint union $\mathcal{A}+\mathcal{B}$. Typically, on the example of NFJ programs, if one wants to use the simpler expression $\Psi = Z + Z_1 \times Z_1 + Z_1 \times Z_1 + Z_1 \times Z_1$ to model the class $\mathcal{F}$, then one needs some extra information at the level of the builder to know from which of the three $Z_1 \times Z_1$ terms each pair is generated. To circumvent this issue, we "tag" each element sampled from a disjoint union with an integer indicating from which operand it is sampled. In the case of the expression Ψ from above, we thus produce either an atom $z^{(1)}$ tagged with 1 or a pair of terms $(x, y)^{(i)}$ tagged with an integer $2 \leq i \leq 4$. At the implementation level, this involves the addition of a new instruction $\texttt{TAG}(i)$. The rule associated with this instruction is also presented in Table 3.6.

Moreover, for the convenience of the user, we also provide a helper function UnionBuilder which hides the plumbing related to these tags by taking a variable number of builder arguments and returning a new builder which accepts a tagged object and calls the right builder on it. More explicitely, UnionBuilder is defined as in Algorithm 44.

Algorithm 44 The UnionBuilder helper

Input: Any number k of builders $f_1, f_2, \dots, f_k$
Output: A builder taking an object tagged with $1 \leq i \leq k$
  function UnionBuilder($f_1, f_2, \dots, f_k$)
    function build(x)
      $y^{(i)} \leftarrow x$ ▹ Match on the tag
      return $f_i(y)$
    return build

For example, in the case of the expression $\Psi = Z + Z_1 \times Z_1 + Z_1 \times Z_1 + Z_1 \times Z_1$ representing NFJ programs, the following builder can be used to generate an expression of X using only the constructors Atom, Par, Seq and Choice:

$$f = \begin{cases} \mathcal{Z} + X \times X + X \times X + X \times X \longrightarrow X \\ x \mapsto \textsc{UnionBuilder}(z \mapsto \texttt{Atom}, f_\parallel, f_;, f_+) \end{cases} \quad \begin{aligned} f_\parallel &= (x, y) \mapsto \texttt{Par}(x, y) \\ f_; &= (x, y) \mapsto \texttt{Seq}(x, y) \\ f_+ &= (x, y) \mapsto \texttt{Choice}(x, y) \end{aligned}$$

Conclusion

Contributions

Throughout this thesis we exposed several contributions revolving around the analysis of the control graph of concurrent programs via analytic combinatorics and random generation. We covered several aspects of this topic, starting from more theoretical results obtained through analytic techniques and leading to various algorithmic applications, especially in the field of *uniform* random generation.

One of our main contributions is the thorough analysis of a class of concurrent programs featuring non-determinism, a fork-join style of synchronisation and loops. To this end, we developed a framework, based on combinatorial specifications and on analytic combinatorics. Using this framework, we obtained two types of results. First, on the analytical side, we established quantitative properties of fork-join programs, related to their number of global choices (in the loop-free fragment) and on their typical number of executions prefixes. Second, and more importantly, we described efficient uniform random samplers of executions and execution prefixes allowing to explore the state-space of programs *without* explicitly constructing it. These algorithms thus provide a tractable way to tackle the state explosion problem.

A second major contribution of this thesis is the introduction and the study of a new class of directed acyclic graphs (directed *ordered* acyclic graphs or DOAGs). This class provides an alternative model to labelled graphs as an approximation of *partial orders*, which are the heart of concurrency since they represent faithfully the control flow of concurrent programs. In order to analyse this class, we use a recursive decomposition scheme of a slightly different kind from the traditional decomposition used in DAG enumeration. Based on this decomposition we are able to sample DOAGs with a given number of vertices and edges uniformly at random and to study a multi-graph variant of this model. Moreover, using a similar approach, we exhibit new counting formulas for the classical model of labelled DAGs allowing to count them by number of vertices and edges without resorting to inclusion-exclusion. This allows to describe a uniform sampler for this class, with control over the number of vertices and edges, which is a new result.

Finally, our work covers various practical aspects of random generation. This includes providing better algorithms for the generation of basic combinatorial objects such as combinations. A key point of this contribution is the use of a more realistic complexity model which allows to see why our new algorithm outperforms the state of the art. In the same line of work, we develop a uniform random sampler of variations with a better complexity than the sampler obtained from the generic framework of recursive method. We also developed a generic and fast Boltzmann sampling library. Our contribution regarding Boltzmann generation is the precise

description of how to implement such samplers in practice, including some optimisations, as well as a formalism to make them easily integrable with other code bases.

Perspectives

Gaining expressiveness by expanding only problematic choices We have made a significant leap forward in terms of expressiveness in Chapter 1, compared to previous works, regarding the uniform random generation of program executions. However, the class of programs we consider is still constrained and, in a sense, too "well-behaved" to model all real-life programs. In fact, a limitation of our approach is that it is a *requirement* for a program construction (or said differently, an operator) to be well-behaved so that we can analyse it using analytic combinatorics.

A direction we wish to explore is to relax the kind of synchronisation we allow to a significantly more expressive model such as e.g. one-safe Petri nets [Pet62]. A Petri net is a bipartite directed graph used to model concurrent systems. It has two types of nodes:

- places, pictured as circles, which may contain any number of tokens;
- and transitions, pictured as rectangles, which represent the actions of the language.

A transition is enabled if all of the places that have an edge to it contain at least one token. And firing an enabled transition consists in removing a token for each of these places and adding one token to all the places to which the transition has an edge. The semantic of a net is expressed as the set of firing sequences that are accessible from an initial marking. One-safe Petri nets are such nets for which it is not possible to have more than one token on every place (from a given initial marking). An example of such net, with its initial marking (only one token in the uppermost place) is given on the left-hand-side of Figure 3.10.

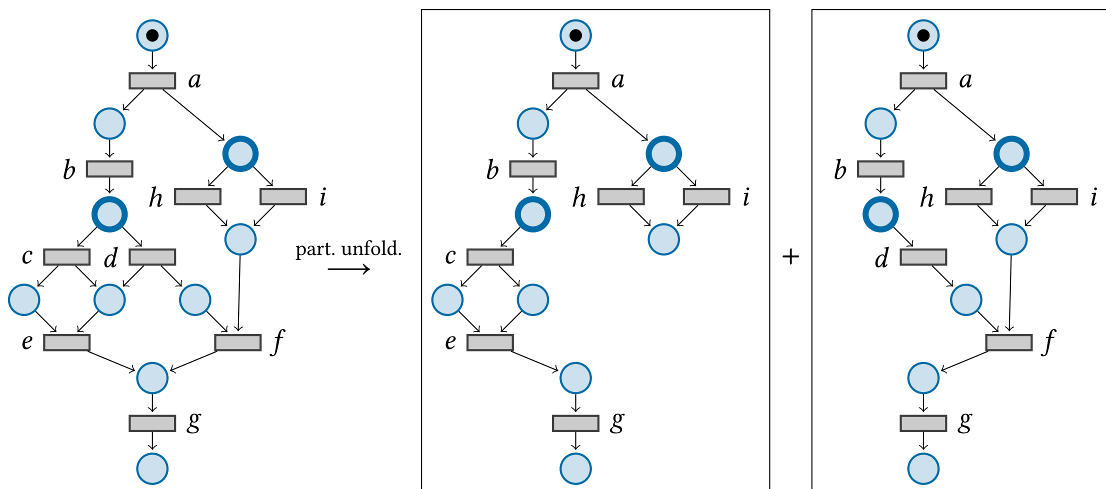


Figure 3.10: An example Petri net and one of its partial unfoldings. The executions of the left-most net are partitioned into two sets, those that fire the transition c and correspond to the first unfolding, and those that fire d and correspond to the second unfolding.

This model is way too expressive to be analysed using the techniques developed in this thesis. Indeed, even in the absence of cycles, one-safe Petri nets are strictly more expressive than partial orders, for which the counting problem is #P-complete and the approach based on analytic combinatorics is not tractable [BDGP19]. However, a possible approach to tackle this kind of programs would be to *expand* (or rather *unfold*, using the Petri net nomenclature) some of the choices of such a net, until the program falls into a class we know how to handle, such as NFJ programs. As an example, consider the program given on the left-hand-side of Figure 3.10. It has two places (indicated by thicker circles) that informally correspond to a choice, because they have multiple outgoing edges. The rightmost choice is not "problematic" in the sense that it does not interfere with any synchronisation. However, the other choice on whether the e and the f transitions can be fired. One can expand this choice, which yields two Petri nets. The first one encodes all the executions that take the c choice, and the second one encodes all the executions that take the d choice. On this example, the resulting two nets both "fall" in the NFJ class in the sense that their executions are in bijections with executions of two NFJ programs. The first net corresponds to the program $a; ((b; c; e; g) \parallel (i + j))$ and the second one corresponds to $a; ((b; d) \parallel (i + j)); f; g$. This allows to count the number of executions of both nets and, based on this information, to sample a uniform execution of the initial net. This is achieved by choosing one of the two expanded nets with the right probability (available by the aforementioned counting) and to sample a uniform execution of this net.

Towards a statistical model checker Another natural direction to explore from here is to apply the techniques developed in the present thesis in a statistical model checker. This implies replacing the atomic actions of the NFJ languages by actual program instructions so that our programs can *compute* and not simply encode a control flow. Our approach would be to give a semantic to these actions for which we can also implement a random sampler. A statistical model checker built on top of this would first sample a uniform *scheduling* of the actions (using the algorithms developed here), then check whether the sampled scheduling is realisable based on the semantics of the actions, and finally sample an execution in the scheduled program. A possible candidate for the language used in place of the atomic actions is given by [GM03] whose semantics can be expressed as a regular language. The advantage of regular languages is that efficient random generation is possible for them, at the cost of explicitly constructing a deterministic finite automata recognising them [BG12].

Exploiting the rationality of the generating function of execution The generating function of the executions of an NFJ program is always rational. Moreover, it is actually possible to recursively compute a rational expression of the form $\frac{A(z)}{B(z)}$, where A and B are polynomials, for this generating function. The recursive rules for computing this expression are clear for all the operators of the NFJ language except for the parallel composition. For instance, if $\frac{A(z)}{B(z)}$ and $\frac{C(z)}{D(z)}$ denote the generating functions of executions of two programs P and Q, then the generating function of the executions of $(P + Q)$ is given by

$$\frac{A(z)D(z) + C(z)B(z)}{B(z)D(z)} - \frac{A(0)C(0)}{B(0)D(0)}$$

and that of $P^\star$ is given by

$$\frac{B(z)}{B(z) - A(z) + \frac{B(z)A(0)}{B(0)}}.$$

The case of the parallel composition requires to compute the coloured product $\frac{A(z)}{B(z)} \circledcirc \frac{C(z)}{D(z)}$ of two rational functions. In fact, the rules developed in Section 1.4.2 (in particular Lemma 4 on page 52 and Equation (1.14) on page 53) are enough to compute such a product provided that a *partial fraction decomposition* of the functions is known. Of course, this method is not constructive since such a decomposition requires to factor polynomials. However, it allows to prove that the zeros of the denominator of the resulting fraction are of the form $(\beta^{-1} + \delta^{-1})^{-1}$ where β is a root of $B(z)$ and δ is a root of $D(z)$ (with known multiplicities). Put differently, this means that the resulting denominator is a divisor of the *resultant* of $y^{d_B}B(1/y)$ and $(z-y)^{d_D}D(1/(z-y))$ with respect to y where d_B and d_D are the degrees of B and D. This resultant can be computed in polynomial time from the expansion of B and D. Finally, an explicit upper bound on the degree of the *numerator* of $\frac{A(z)}{B(z)} \circledcirc \frac{C(z)}{D(z)}$ can be given so that its (irreducible) fraction decomposition can be obtained in polynomial time in the size of $A(z)$, $B(z)$, $C(z)$ and $D(z)$.

This approach has a major drawback however. The degree of the denominator of $\frac{A(z)}{B(z)} \circledcirc \frac{C(z)}{D(z)}$ is the product of the degrees of B and D (modulo cancellations). Hence this degree grows quickly with the number of coloured products. As a consequence, computing a rational expression for the generating function of the executions of a program is actually exponential in the size of the program. On the other hand, for small programs for which this remains tractable, having a rational expression has important advantages. In particular, this gives a linear recurrence relation on the number of executions of size n of the program which could potentially accelerate significantly the pre-computation phase of our algorithms. An important question left open here is: to what extent is this approach tractable in practice?

Using the floating point optimisation At the moment, the bottleneck of our random generator of executions is the pre-computation of the coefficients of the generating functions. The complexity of this pre-processing is due to two factors. First, it requires to perform a large number of polynomial multiplications which is a costly operation. The ideas exposed above regarding the rationality of the generating functions at play addresses this issue by suggesting a cheaper approach, though it is only tractable for small-size programs. The second major factor of complexity is due to big integer arithmetic which becomes more and more costly as the size of the integers grow. A way to mitigate this cost has been proposed in [DZ99] where the idea is to resort to certified floating point arithmetic. In fact, two approaches are proposed.

A first possibility to make the random generation more efficient is to only use floating point numbers for manipulating and storing the coefficients of the generating functions. The authors of [DZ99] provide theoretical error bounds allowing to quantify the default of uniformity of this approach. Moreover, they also explain how to compute more precise error bounds during the generation and, according to their experiments, these bounds seem much better than the theoretical ones. This approach provides a trade-off between uniformity and performance, which can be reasonable in practice.

Another approach proposed by [DZ99] is to remain perfectly uniform by using the (theoretical or dynamically computed) errors bounds to decide which algorithm to use for the gen-

eration. The key idea here is that most of the time, approximate values are enough for making exact choices. In the few cases where this is not the case, the solution consists in either resorting to the exact algorithm using integer arithmetic, or to refine our approximations by doubling the precision. Let us illustrate this concretely on an example. Most of the time, the randomness of our algorithms lies in the generation of a Bernoulli variable. Say one needs to generate a Bernoulli variable $X \longleftarrow \textsc{Bern}(p)$ of parameter p and that one only has an approximation $\bar{p}$ of p such that $|p - \bar{p}| < \epsilon$. One solution consists in drawing a uniform real number $U \in [0;1]$ (it can be represented by a lazy infinite sequence of bits). If $U < \bar{p} - \epsilon$, then $U < p$ and $X = \text{true}$. Similarly, if $U > \bar{p} + \epsilon$, then $U > p$ and we have $X = \text{false}$. The only problematic cases are those where $|U - \bar{p}| < \epsilon$. In that case, we need to compute a better approximation of p to be able to conclude.

Note that in both approaches, the sizes of the numbers at play require us to use a special representation with at least a larger exponent than the one provided by default in most programming languages. This can be achieved, for instance, using the *extendable* precision format specified by the IEEE 754 standard [1].

More efficient and precise analysis of the isotropic method In Section 1.4.4 we compared the coverage provided by our uniform sampler of prefixes with that of the isotropic method for a simple program. Unfortunately we could only do this for a program of small size because of the prohibitive cost of computing the coverage rate of the isotropic method. This raises two questions that have been left open.

First, is there a more efficient way of assessing the coverage of the isotropic method? The formula we used is due to [FGT92] and is rather general. It seems reasonable to hope for a simpler formula in our context where the probabilities of the prefixes are obtained by decomposing a tree-like object.

Second, and as an alternative to a more efficient algorithm, theoretical bounds on the cost of the isotropic method would be rather helpful too. In particular, a lower bound on the time necessary to achieve a given coverage would allow to invalidate the isotropic method on more than a simple example. This would reinforce the need for samplers with a *controlled* probability distribution, and in particular the *uniform* distribution.

Typical properties of DOAGs For now, we have focused mostly on the uniform random sampling problem for the class of DOAGs. It would be interesting to study some parameters of these graphs in average such as the distribution of their degrees, the height, etc. In particular, the quantity $\sum_v d_v^2$, where v ranges over the vertices of a graph and d_v denotes the out-degree of v, is of particular interest since it appears in the complexity of our random sampler of DOAGs in Section 2.2.3.

Another related question is to know the number of sparse DOAGs (say when $m = O(n)$) and their behaviour. A classical question in graph and digraph enumeration is to describe the phase transition between disconnectedness and connectedness (see [Pan+20] for the digraph case). A first step towards this, in our context, would be to define a more general class of DOAGs that are not necessarily (weakly) connected and to study the proportion of (weakly) connected DOAGs as a function of the number m of edges.

Asymptotic number of DOAGs with n vertices We stated a conjecture on the number $D_{n,1}$ of DOAGs with n vertices, one source and any number of edges in Section 2.2.4 (Conjecture 1 on page 74). We recall here the recurrence satisfied by the sequence $D_{n,k}$ of DOAGs with n vertices and k sources for convenience:

$$D_{n,k} = \sum_{s \geq 0} D_{n-1,k-1+s} \cdot \gamma(n-k-s, s) \qquad (\text{for } k > 0 \text{ and } n > 1) \tag{3.12}$$

where

$$\gamma(a, b) = \mathbb{1}_{\{b \neq 0\}} + \sum_{i > 0} \binom{b+i}{b} \binom{a}{i} i!$$

This conjecture is based on the upper and lower bounds we established on $D_{n,1}$ in Section 2.2.4. In fact, similar bounds can be found for $D_{n,k}$ for any value of k and with a similar precision (of the order of $O(n)$) provided that k remains small. Moreover, we can show that $\gamma(a, b) = \frac{(a+b)!}{b!} \left(\exp\left(\frac{b}{a+b}\right) + o(1)\right)$ when $\min(a, b) \longrightarrow \infty$. A natural lead to proving Conjecture 1 would be to plug these approximations into (3.12) and try to refine them.

A similar approach has already been used in [BGGW20] to study a family of weakly increasing trees. The idea is to identify the terms that contribute the most to the sum, normalise the sum by the expected first order (here ¡$(n-1)! e^{n-1} = \prod_{k=1}^{n-1} e \cdot k!$) and simplify the summand so as to make a functional equation or a differential equation appear. If the approximations made in the process are precise enough, this yields upper and lower bounds on the initial sequence allowing to conclude.

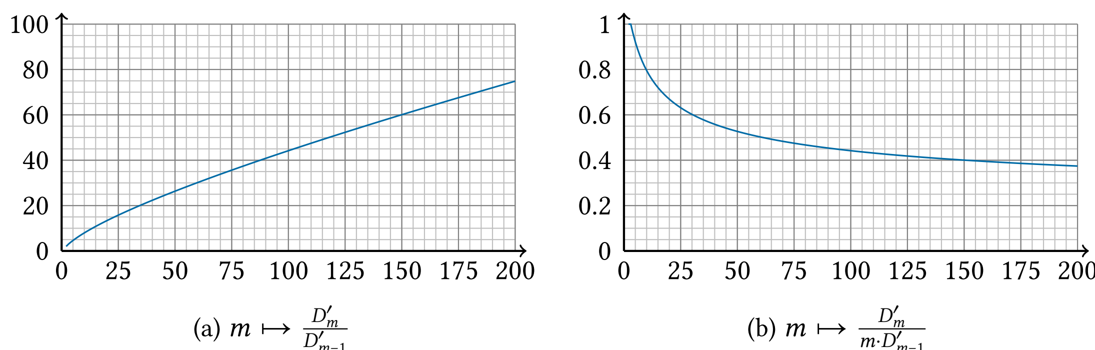
Studying the asymptotic number of DOAMGs Understanding, even roughly, the asymptotic behaviour of the sequence $(D_{n,1})_{n>0}$ counting the number of DOAGs with n vertices had led us to a fast pre-computation-free sampler for these objects. A natural question, which is left open by Section 2.4 is: what can be say about the asymptotic number of DOAMGs (Directed Acyclic Ordered *Multi*-Graphs)? As a starting point in this direction, we would like to study the sequence $m \mapsto D'_m = \sum_n D'_{n,m,1}$ counting DOAMGs with m edges and any number of vertices.

As a preparatory experiment, we plotted the ratios $\frac{D'_m}{D'_{m-1}}$ and $\frac{D'_m}{m \cdot D'_{m-1}}$ on Figure 3.11 on the next page. It is typical for sequences counting *unlabelled* objects that the former ratio converges to a constant whereas, for *labelled* objects, the latter generally tends to a constant. While the sequence D'_m clearly does not belong to the unlabelled world according to the plot, it is not obvious either whether the second ratio converges or not. A convergence to 0 would indicate a rather unusual behaviour of the sequence. A more likely scenario is that this sequence shares a common property with the sequence counting the number of compressed binary trees: having a stretched exponential [EFW21]. More eloquently having an asymptotic behaviour of the form

$$D'_m = C \cdot n^{\alpha} r^n \exp(\lambda n^{\beta}) n! \qquad (\text{with } 0 < \beta < 1)$$

would explain this slow convergence.

One strong argument in favour of this hypothesis is that the objects counted by D'_m and in [EFW21] are very similar: compacted binary trees are DOAMGs with some extra constraints, in particular degree constraints. Moreover, the bijection we established in Section 2.4.3 on page 80 involving decorated paths have similarities too with the bijection used in [EFW21] to



(a) $m \mapsto \frac{D'_m}{D'_{m-1}}$ (b) $m \mapsto \frac{D'_m}{m \cdot D'_{m-1}}$

Figure 3.11: Preparatory experimental study of the sequence D'_m.

simplify the counting problem. We believe this bijection is a good starting point to study D'_m, notably because:

- counting pairs of non-intersecting (non-decorated) walks is tractable since they admit a context-free grammar;
- the authors of [EFW21] have developed heuristics and techniques to study horizontally decorated walks.

The problems thus boils down to: can we combine the two techniques to study D'_m? This is at the moment an open question.

A more efficient sampler of DOAMGs? In Section 2.2.5 on page 74 we have developed a fast, pre-computation-free, uniform sampler of DOAGs with n vertices. This key idea was to consider its transition matrix and sample a product of $(n - 1)$ variations (each representing one line of the matrix) until they form a valid DOAG transition matrix. Combined with an early rejection procedure, this approach has proved to be efficient in practice.

A natural question we wish to investigate is: can we do the same for DOAMGs? While the notion of transition matrix does not neatly generalise to multi-graphs, the bijection we established in Section 2.4.3 on page 80 with pairs of non-intersecting decorated paths seems to be a good candidate for a similar sampler. As a future work, we want to develop a uniform sampler of horizontally decorated path and to combine it with a uniform sampler of non-decorated path to sample DOAMGs. Preventing the two paths from intersecting can be done using rejection and we hope for this method to be efficient if the rejection is made as soon as possible during the generation of the two paths.

Adding pointing (and enhanced pointing?) to usainboltz Although our usainboltz library [6] has already reached a usable stage for basic algebraic use cases, we wish to extend its feature set. In particular, at the moment the preferred way to achieve linearity in usainboltz while sampling structures described by an algebraic grammar is using singular sampling. The pointing technique described in [DFLS04] has not yet been automated so that a user wishing to use it has to derive the grammar manually. An important future work is to automate this derivation and the reconstruction of the objects after the pointing has taken place.

Moreover, another potential application of pointing, which to our knowledge has never been mentioned before, is to make the SEQ operator appear as the outermost element of the grammar so that the "leap frogging" technique from [DFLS04, page 613] can apply. This is better explained with an example.

Consider the grammar of binary trees $\mathcal{B} = \mathcal{E} + \mathcal{Z} \times \mathcal{B} \times \mathcal{B}$. The grammar for pointed binary trees $\mathcal{B}^\bullet$ can be obtained using the recursive rules from [DFLS04, page 610], which yields:

$$\mathcal{B}^\bullet = \mathcal{Z}^\bullet \times \mathcal{B} \times \mathcal{B} + \mathcal{Z} \times \mathcal{B}^\bullet \times \mathcal{B} + \mathcal{Z} \times \mathcal{B} \times \mathcal{B}^\bullet \tag{3.13}$$

The combinatorial interpretation of this specification is that a binary tree with one pointed (or "distinguished") node is either:

- a pointed root ($\mathcal{Z}^\bullet$) with two regular trees ($\mathcal{B}$) as children;
- a regular root ($\mathcal{Z}$) with a pointed tree ($\mathcal{B}^\bullet$) on the left and a regular tree ($\mathcal{B}$) on the right;
- a regular root ($\mathcal{Z}$) with a regular tree ($\mathcal{B}$) on the left and a pointed tree ($\mathcal{B}^\bullet$) on the right.

In [DFLS04], the authors prove that using this grammar to generated binary trees by forgetting which node was pointed in the end yields better performance than sampling trees from $\mathcal{B} = \mathcal{E} + \mathcal{Z} \times \mathcal{B} \times \mathcal{B}$ directly.

There is actually a second way of specifying $\mathcal{B}^\bullet$ which allows to use the "leap-frogging" technique and thus to potentially accelerate the generation further. Since the right-hand-side of (3.13) is affine in $\mathcal{B}^\bullet$, it can be modified as follows:

$$\begin{aligned} \mathcal{B}^\bullet &= \mathcal{Z}^\bullet \times \mathcal{B} \times \mathcal{B} \times \textsc{Seq}(\mathcal{Z} \times \mathcal{B} + \mathcal{Z} \times \mathcal{B}) \\ &= \mathcal{B} \times \textsc{Seq}(\mathcal{Z} \times \mathcal{B} + \mathcal{Z} \times \mathcal{B}) && \text{(since } \mathcal{Z}^\bullet \text{ is in bijection with } \mathcal{Z}\text{)} \end{aligned}$$

Thus, generating a pointed binary tree whose size is in some size interval $I(n, \epsilon) = [(1 - \epsilon)n; (1 + \epsilon)n]$ can be achieved by the following algorithm:

1. sample a first tree of size $s_1 \leq (1 + \epsilon)n$;
2. sample a sequence of trees from $(\mathcal{Z} \times \mathcal{B} + \mathcal{Z} \times \mathcal{B})$ using the leap-frogging algorithm (more eloquently, keep sampling elements of the sequence until their total size reaches $(1 - \epsilon)n - s_1$);
3. at the end, check whether the total size is below $(1 + \epsilon)n$ and start over if not.

The trick we used to make a sequence appear in the specification of $\mathcal{B}^\bullet$ actually applies to other classes of trees. It is actually known from the functional programming community as a technique to define *zippers* [Hue97]. Zippers are efficient data structures for representing trees where the focus is set of one particular sub-tree and where the basic operations of moving the focus to neighbouring sub-trees is cheap. The idea consists in turning the tree "inside out" so as to have access to (1) the sub-tree of interest and (2) the sequence of ancestors of that sub-tree starting from the closest one. The parallel between the two worlds is as follows.

- The pointing operation from analytic combinatorics corresponds to setting the focus on one sub-tree of a tree-like structure.
- And the "inside-out" representation used for zippers corresponds to factoring the occurrences of the pointed class as a sequence as we did for binary trees above.

We wish to implement this idea as part of usainboltz. However, work remains to be done to establish whether the leap-frogging technique actually out-performs basic pointing or not and to quantify it. Moreover, we need to make this process automatic and invisible to the user.

Handling autonomous differential equations in usainboltz We only covered unlabelled Boltzmann sampling in Section 3.4 on page 118 but our usainboltz library is also able to handle labelled specifications. One of the lesser-known operators of labelled specifications is the *boxed* product $^{\square}\star$. Given two labelled combinatorial classes $\mathcal{A}$ and $\mathcal{B}$, the class $\mathcal{A}^{\square}\star\mathcal{B}$ is the sub-set of the elements of $\mathcal{A} \star \mathcal{B}$ such that the smallest label (1) is on the left object. This operator also has an interpretation in terms of generating functions since the *exponential* generating function (EGF) of $\mathcal{A}^{\square}\star\mathcal{B}$ is given by $\int_0^z A'(t)B(t)\mathrm{d}t$ where $A(t)$ and $B(t)$ represent the EGFs of $\mathcal{A}$ and $\mathcal{B}$. More details on this operator can be found in [FS09, page 139].

It has been shown in [BRS12] that the first-order differential specifications arising from the use of the boxed product can be integrated in the Boltzmann framework. In fact, the approach developed there can be applied to higher order differential equations too [Die17, page 60]. One major inconvenience of this approach however is that it is computationally demanding on the oracle side as one has to compute integrals with different bounds at each recursive call. However, both [BRS12] and [Die17] note that, in the autonomous case, that is when the differential equation is of the form $f'(z) = T(f(z))$, these computations are not required.

We wish to add support for autonomous differential specifications in usainboltz, which implies to formally describe the general case and to handle the labelling of such structures efficiently.

Academic references

[AM15] Samy Abbes and Jean Mairesse. "Uniform Generation in Trace Monoids". In: *40th International Symposium on Mathematical Fundations of Computer Science (MFCS)*. Aug. 2015, pages 63–75.

[BBD18] Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. "Polynomial tuning of multiparametric combinatorial samplers". In: *2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. SIAM. 2018, pages 92–106.

[BDGP17a] Olivier Bodini, Matthieu Dien, Antoine Genitrini, and Frédéric Peschanski. "Entropic Uniform Sampling of Linear Extensions in Series-Parallel Posets". In: *12th International Computer Science Symposium in Russia (CSR)*. 2017, pages 71–84. DOI: 10.1007/978-3-319-58747-9_9.

[BDGP17b] Olivier Bodini, Matthieu Dien, Antoine Genitrini, and Peschanski Peschanski. "The Ordered and Colored Products in Analytic Combinatorics: Application to the Quantitative Study of Synchronizations in Concurrent Processes". In: *2017 Proceedings of the Fourteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. SIAM. 2017, pages 16–30.

[BDGP19] Olivier Bodini, Matthieu Dien, Antoine Genitrini, and Frédéric Peschanski. "The Combinatorics of Barrier Synchronization". In: *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer. 2019, pages 386–405.

[BFKV11] Manuel Bodirsky, Éric Fusy, Mihyun Kang, and Stefan Vigerske. "Boltzmann samplers, Pólya theory, and cycle pointing". In: *SIAM Journal on Computing* 40.3 (2011), pages 721–769.

[BG12] Olivier Bernardi and Omer Giménez. "A linear algorithm for the random sampling from regular languages". In: *Algorithmica* 62.1 (2012), pages 130–145.

[BGGW20] Olivier Bodini, Antine Genitrini, Bernhard Gittenberger, and Stephan Wagner. "On the number of increasing trees with label repetitions". In: *Discrete Mathematics* 343.8 (2020), page 111722. ISSN: 0012-365X. DOI: https://doi.org/10.1016/j.disc.2019.111722.

[BGP12] Olivier Bodini, Antoine Genitrini, and Frédéric Peschanski. "Enumeration and Random Generation of Concurrent Computations". In: *23rd International Meeting on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms*. Montreal, Canada, June 2012, pages 83–96. URL: http://www.dmtcs.org/dmtcs-ojs/index.php/proceedings/issue/view/125.

[BGP13] Olivier Bodini, Antoine Genitrini, and Frédéric Peschanski. "The Combinatorics of Non-determinism". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013)*. Volume 24. Leibniz International Proceedings in Informatics (LIPIcs). Guwahati, India, 2013, pages 425–436.

[BGP16] Olivier Bodini, Antoine Genitrini, and Frédéric Peschanski. "A Quantitative Study of Pure Parallel Processes". In: *Electronic Journal of Combinatorics* 23.P1.11 (1 2016).

[BGR15] Olivier Bodini, Antoine Genitrini, and Nicolas Rolin. "Pointed versus Singular Boltzmann Samplers: a Comparative Analysis". In: *PUre Mathematics and Applications* 25.2 (2015), pages 115–131.

[BL77] Bill Paul Buckles and Matthew Lybanon. "Algorithm 515: Generation of a vector from the lexicographical index [G6]". In: *ACM Transactions on Mathematical Software* 3.2 (1977), pages 180–182.

[BMS17] Nicolas Basset, Jean Mairesse, and Michèle Soria. "Uniform sampling for networks of automata". In: *28th International Conference on Concurrency Theory (CONCUR 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[Bos+17] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Grégoire Lecerf, Bruno Salvy, and Éric Schost. *Algorithmes efficaces en calcul formel*. Published by the authors, 2017.

[BRRW86] Edward Anton Bender, Lawrence Bruce Richmond, Robert William Robinson, and Nicholas Charles Wormald. "The asymptotic number of acyclic digraphs". In: *Combinatorica* 6.1 (1986), pages 15–22.

[BRS12] Olivier Bodini, Olivier Roussel, and Michèle Soria. "Boltzmann samplers for first-order differential specifications". In: *Discrete Applied Mathematics* 160.18 (Dec. 2012), pages 2563–2572.

[But15] Brian Butler. *Function kSubsetLexUnrank, MATLAB Central File Exchange*. 2015.

[CEH19] Louis-Claude Canon, Mohamad El Sayah, and Pierre-Cyrille Héam. "A Comparison of Random Task Graph Generation Methods for Scheduling Problems". In: *European Conference on Parallel Processing*. Volume 11725. LNCS. Springer. 2019, pages 61–73. DOI: 10.1007/978-3-030-29400-7_5.

[Cor+10] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. "Random graph generation for scheduling simulations". In: *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST. 2010, page 10.

[Den+12] Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. "Coverage-biased random exploration of large models and application to testing". In: *International Journal on Software Tools for Technology Transfer* 14.1 (2012), pages 73–93.

[DFLS04] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. "Boltzmann samplers for the random generation of combinatorial structures". In: *Combinatorics, Probability & Computing* 13.4-5 (2004), pages 577–625.

[Die17] Matthieu Dien. “Processus concurrents et combinatoire des structures croissantes : analyse quantitative et algorithmes de génération aléatoire”. PhD thesis. Université Pierre et Marie Curie - Paris VI, Sept. 2017. URL: http://www.theses.fr/2017PA066210.

[DPRS10] Alexis Darrasse, Konstantinos Panagiotou, Olivier Roussel, and Michèle Soria. “Boltzmann generation for regular languages with shuffle”. In: *GASCOM 2010 - Conference on random generation of combinatorial structures*. Montréal, Canada, Sept. 2010.

[Drm09] Michael Drmota. *Random Trees: An Interplay between Combinatorics and Probability*. Springer-Verlag, 2009.

[Dur64] Richard Durstenfeld. “Algorithm 235: Random Permutation”. In: *Communications of the ACM* 7.7 (1964), page 420.

[DZ99] Alain Denise and Paul Zimmermann. “Uniform random generation of decomposable structures using floating-point arithmetic”. In: *Theoretical Computer Science* 218.2 (1999), pages 233–248. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(98)00323-5. URL: https://www.sciencedirect.com/science/article/pii/S0304397598003235.

[EFW21] Andrew Elvey Price, Wenjie Fang, and Michael Wallner. “Compacted binary trees admit a stretched exponential”. In: *Journal of Combinatorial Theory, Series A* 177 (2021), page 105306. ISSN: 0097-3165. DOI: https://doi.org/10.1016/j.jcta.2020.105306. URL: https://www.sciencedirect.com/science/article/pii/S0097316520300984.

[Er85] M. C. Er. “Lexicographic ordering, ranking and unranking of combinations”. In: *International Journal of Computer Mathematics* 17.3-4 (1985), pages 277–283. DOI: 10.1080/00207168508803468.

[Ers58] Andrey Petrovych Ershov. “On Programming of Arithmetic Operations”. In: *Communications of the ACM* 1.8 (Aug. 1958), pages 3–6. ISSN: 0001-0782.

[Fel50] William Feller. *An Introduction to Probability Theory and its Applications*. Volume 1. John Wiley and Sons, Inc., New York, 1950.

[FFP07] Philippe Flajolet, Éric Fusy, and Carine Pivoteau. “Boltzmann sampling of unlabelled structures”. In: *2007 Proceedings of the Fourth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. SIAM. 2007, pages 201–211.

[FGT92] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. “Birthday paradox, coupon collectors, caching algorithms and self-organizing search”. In: *Discrete Applied Mathematics* 39.3 (1992), pages 207–229.

[FN13] Sven de Felice and Cyril Nicaud. “Random Generation of Deterministic Acyclic Automata Using the Recursive Method”. In: *8th International Computer Science Symposium in Russia, CSR’13*. Volume 7913. LNCS. Springer, 2013, pages 88–99. DOI: 10.1007/978-3-642-38536-0_8. URL: https://doi.org/10.1007/978-3-642-38536-0%5C_8.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009, pages I–XIII, 1–810. ISBN: 978-0-521-89806-5.

[FY48] Ronald Aylmer Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. London: Oliver and Boyd, 1948.

[FZV94] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. "A calculus for the random generation of labelled combinatorial structures". In: *Theoretical Computer Science* 132.1-2 (1994), pages 1–35.

[Ges95] Ira Martin Gessel. "Enumerative applications of a decomposition for graphs and digraphs". In: *Discrete Mathematics* 139.1 (1995), pages 257–271. ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(94)00135-6.

[Ges96] Ira Martin Gessel. "Counting acyclic digraphs by sources and sinks". In: *Discrete Mathematics* 160.1 (1996), pages 253–258. ISSN: 0012-365X.

[GM03] Dan Razvan Ghica and Guy McCusker. "The regular-language semantics of second-order Idealized Algol". In: *Theoretical Computer Science* 309.1-3 (2003), pages 469–502.

[Got74] Eiichi Goto. *Monocopy and associative algorithms in an extended lisp*. Technical report. University of Tokyo, 1974.

[GP21] Antoine Genitrini and Martin Pépin. "Lexicographic unranking of combinations revisited". In: *Algorithms* 14.3 (2021), page 97.

[GPP20] Antoine Genitrini, Martin Pépin, and Frédéric Peschanski. "Statistical Analysis of Non-Deterministic Fork-Join Processes". In: *International Colloquium on Theoretical Aspects of Computing*. Springer. 2020, pages 83–102.

[GPV21] Antoine Genitrini, Martin Pépin, and Alfredo Viola. "Unlabelled ordered DAGs and labelled DAGs: constructive enumeration and uniform random sampling". In: *XI Latin and American Algorithms, Graphs and Optimization Symposium*. Eslevier. 2021.

[GS05] Radu Grose and Scott Allen Smolka. "Monte Carlo Model Checking." In: *TACAS*. Volume 3440. Springer. 2005, pages 271–286.

[Hoa78] Charles Antony Richard Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (1978), pages 666–677.

[Hue97] Gérard Huet. "Functional pearl: The Zipper". In: *Journal of functional programming* 7.5 (1997), pages 549–554.

[Izq59] Sebastián Izquierdo. *Pharus scientiarum*. Volume 2. sumptibus Claudii Bourgeat & Mich. Lietard, 1659.

[JCB00] Aubin Jarry, Henri Casanova, and Francine Berman. "Dagsim: A simulator for dag scheduling algorithms". In: *École Normale Supérieure de Lyon, Research Report No* 46 (2000).

[KM15] Jack Kuipers and Giusi Moffa. "Uniform random generation of large acyclic digraphs". In: *Statistics and Computing* 25.2 (2015), pages 227–242.

[KMM02] Daniel Krob, Jean Mairesse, and Ioannis Michos. "On the Average Parallelism in Trace Monoids". In: *Annual Symposium on Theoretical Aspects of Computer Science.* Springer. 2002, pages 477–488.

[Knu97] Donald Ervin Knuth. *The art of computer programming, volume 2 seminumerical algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[KS99] Donald Lawson Kreher and Douglas Robert Stinson. *Combinatorial Algorithms: generation, enumeration, and search.* CRC Press, 1999.

[Lai88] Charles-Ange Laisant. "Sur la numération factorielle, application aux permutations". In: *Bulletin de la Société Mathématique de France* 16 (1888), pages 176–183.

[McC04] James McCaffrey. "Generating the mth Lexicographical Element of a Mathematical Combination". In: *MSDN* (2004). URL: http://docs.microsoft.com/en-us/previous-versions/visualstudio/aa289166(v=vs.70).

[MDB01] Guy Melançon, Isabelle Dutour, and Mireille Bousquet-Mélou. "Random Generation of Directed Acyclic Graphs". In: *Electronic Notes in Discrete Mathematics* 10 (2001), pages 202–207. DOI: 10.1016/S1571-0653(04)00394-4. URL: https://doi.org/10.1016/S1571-0653(04)00394-4.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. "Functional programming with bananas, lenses, envelopes and barbed wire". In: *Functional Programming Languages and Computer Architecture.* Edited by John Hughes. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pages 124–144. ISBN: 978-3-540-47599-6.

[Mil80] Robin Milner. "A calculus of communicating systems". In: *Lecture Notes in Computer Science* 92 (1980).

[MM04] Conrado Martínez and Xavier Molinero. "An Experimental Study of Unranking Algorithms". In: *Experimental and Efficient Algorithms.* Edited by Celso C. Ribeiro and Simone L. Martins. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pages 326–340. ISBN: 978-3-540-24838-5.

[Mol05] Xavier Molinero. "Ordered generation of classes of combinatorial structures". PhD thesis. Universitat Politècnica de Catalunya, Oct. 2005.

[NW78] Albert Nijenhuis and Herbert Wilf. *Combinatorial Algorithms: For Computers and Hard Calculators.* 2nd. USA: Academic Press, Inc., 1978. ISBN: 0125192606.

[Oud+11] Johan Oudinet, Alain Denise, Marie-Claude Gaudel, Richard Lassaigne, and Sylvain Peyronnet. "Uniform monte-carlo model checking". In: *International Conference on Fundamental Approaches to Software Engineering.* Springer. 2011, pages 127–140.

[Pan+20] Élie de Panafieu, Sergey Dovgal, Dimbinaina Ralaivaosaona, Vonjy Rasendrahasina, and Stephan Wagner. *The birth of the strong components.* 2020. arXiv: 2009.12127 [math.CO].

[Pas87] Ernesto Pascal. "Sopra una formula numerica". In: *Giornale di Matematiche* 25 (1887), pages 45–49.

[PD19] Élie de Panafieu and Sergey Dovgal. "Symbolic method and directed graph enumeration". In: *Acta Mathematica Universitatis Comenianae* 88.3 (2019), pages 989–996. ISSN: 0862-9544. URL: http://www.iam.fmph.uniba.sk/amuc/ojs/index.php/amuc/article/view/1308.

[PD20] Élie de Panafieu and Sergey Dovgal. *Counting directed acyclic and elementary digraphs*. 2020. arXiv: 2001.08659 [math.CO].

[Pet62] Carl Adam Petri. "Fundamentals of a Theory of Asynchronous Information Flow". In: *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. North-Holland, 1962, pages 386–390.

[Piv08] Carine Pivoteau. "Génération aléatoire de structures combinatoires: méthode de Boltzmann effective". PhD thesis. Université Pierre et Marie Curie, Dec. 2008.

[Pól+64] George Pólya, Derrick Henry Lehmer, Montgomery Jr. Phister, John Riordan, Elliott Waters Montroll, Nicolaas Govert de Bruijn, Frank Harary, Richard Bellman, Robert Kalaba, Edwin L. Peterson, Leo Breiman, Albert William Tucker, Edwin Ford Beckenbach, Marshall Jr. Hall, Jacob Wolfowitz, Charles Brown Tompkins, Kenneth N. Trueblood, George Gamow, and Hermann Weyl. *Applied Combinatorial Mathematics*. John Wiley and Sons, inc., New York • London • Sydney, 1964.

[PR12] George Pólya and Ronald C. Read. *Combinatorial enumeration of groups, graphs, and chemical compounds*. Springer Science & Business Media, 2012.

[PSS12] Carine Pivoteau, Bruno Salvy, and Michele Soria. "Algorithms for combinatorial structures: Well-founded systems and Newton iterations". In: *Journal of Combinatorial Theory, Series A* 119.8 (2012), pages 1711–1773.

[Rob73] Robert William Robinson. "Counting labeled acyclic digraphs". In: *New Directions in the Theory of Graphs* (1973), pages 239–273.

[Rob77] Robert William Robinson. "Counting unlabeled acyclic digraphs". In: *Combinatorial Mathematics V*. Lecture Notes in Mathematics. Springer, 1977, pages 28–43.

[Rus03] Frank Ruskey. *Combinatorial generation*. Preliminary working draft. University of Victoria, Victoria, BC, Canada, 2003.

[SP95] Neil James Alexander Sloane and Simon Plouffe. *The encyclopedia of integer sequences*. Academic Press, 1995.

[SS71] Arnold Schönhage and Volker Strassen. "Schnelle multiplikation grosser zahlen". In: *Computing* 7.3 (1971), pages 281–292.

[Sta73] Richard Peter Stanley. "Acyclic orientations of graphs". In: *Discrete Mathematics* 5.2 (1973), pages 171–178.

[Sta86] Richard Peter Stanley. *Enumerative Combinatorics*. Volume 1. The Wadsworth and Brooks/Cole Mathematics Series. Springer, 1986. ISBN: 978-1-4615-9765-0. DOI: 10.1007/978-1-4615-9763-6.

[VG13] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.

Technical references

[1] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pages 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[2] Maciej Bendkowski. *boltzmann-brain: Analytic sampler compiler for combinatorial systems.* URL: https://github.com/maciej-bendkowski/boltzmann-brain.

[3] Maciej Bendkowski. *paganini: Multiparametric tuner for combinatorial specifications.* URL: https://github.com/maciej-bendkowski/paganini.

[4] The Sage Developers. *SageMath, the Sage Mathematics Software System.* 2021. URL: https://www.sagemath.org.

[5] Matthieu Dien, Antoine Genitrini, Marwan Ghanem, Martin Pépin, Frédéric Peschanski, and Xuming Zhan. *arbogen: a fast uniform random tree generator.* URL: https://github.com/fredokun/arbogen.

[6] Matthieu Dien and Martin Pépin. *usainboltz: Uniform SAmplINg with BOLTZmann.* URL: https://gitlab.com/ParComb/usain-boltz.

[7] Torbjörn Granlund et al. *GMP: the Gnu Multi precision arithmetic Library.* URL: https://gmplib.org.

[8] Kenneth Geisshirt, Emanuele Zattin, Aske Olsson, and Rasmus Voss. *Git Version Control Cookbook: Leverage Version Control to Transform Your Development Workflow and Boost Productivity, 2nd Edition.* Packt Publishing, 2018. ISBN: 1789137543.

[9] William Hart, Fredrik Johansson, and Sebastian Pancratz. *FLINT: Fast Library for Number Theory.* Version 2.5.2, http://flintlib.org/authors.html. 2013.

[10] Waseda University Kasahara Laboratory. *STG: standard graph task set.* URL: http://www.kasahara.cs.waseda.ac.jp/schedule/ (visited on 06/02/2021).

[11] Bruno Salvy, Carine Pivoteau, and Pablo Rotondo. *NewtonGF.* URL: https://perso.ens-lyon.fr/bruno.salvy/software/the-newtongf-package/.