Uniform SAmplINg with BOLTZmann*

Matthieu Dien[1] and Martin Pépin[2][0000−0003−1892−3017]

[1] Normandie Université, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France, matthieu.dien@unicaen.fr
[2] Université Paris Cité, IRIF, 75205 Paris Cedex 13, France, martin.pepin@irif.fr

Abstract. Usain Boltz is a fast Python library for the uniform random generation of tree-like structures. It allows the user to specify both (1) the data structure they wish to sample, using simple combinators similar to those of context-free grammars, and (2) their memory representation. The underlying algorithms are optimised Boltzmann samplers allowing to get approximate-size uniform random generation in linear time. Experimental results show that Usain Boltz matches the performance of the experimental Arbogen package for OCaml, and out-performs the Boltzmann brain Haskell library, while being easier to integrate into existing scientific tools such as Sagemath.

Keywords: Uniform random sampling · Boltzmann model · Algorithms

1 Introduction

Discrete structures are often used in sciences to model a concept or an object belonging to the original domain of study. For example, tree structures represent lineage relations between people (history), words, languages (linguistics), or genes (biology); and they also structure information (tree diagrams in probabilities or data structures in computer science). In order to understand the underlying patterns governing the shape of the structures, random generation has proven to be a useful experimentation tool. It is also a basic block of other algorithms. For example, in the context of software engineering, it allows to test the scalability of applications [19], to perform *unit tests* [8,7], or to study the asymptotic complexity of algorithms. In bioinformatics, the random generation of RNA secondary structures is also used to design RNA molecules for nanomaterials or therapeutics [22].

In these examples, the goal is to simulate the uniform distribution: each outcome (of the same size) has the same probability. Various generic approaches have been developed to devise uniform samplers. Notable examples include Monte Carlo techniques, based on Markov chains [16,17,23]; the famous recursive method from Nijenhuis and Wilf [20,15,18], which applies to any data structure

* This research was partially supported by the ANR PPS project ANR-19-CE48-0014 and the "DYNNET" project, co-funded by the Normandy County Council and the European Union in the framework of the ERDF-ESF operational program 2014-2020.

admitting a combinatorial decomposition; and finally the *Boltzmann method* [11], which can generically sample any data structure admiting a combinatorial *specification* in the sense of [12].

This last method generally out-performs the others by achieving *linear* time sampling if one can tolerate a small imprecision in the size of the generated object. This is why we opted for Boltzmann sampling in our library Usain Boltz. Our implementation[3] has fast `C++` routines in its core and is exposed as a Python [24] library so that it integrates well into the rich software environment that exists in Python for scientific computing, notably thanks to Sagemath [26]. Our implementation is open-source (GPLv3 license) to support the open science dynamics and the free software movement.

The rest of this paper is organised as follows. Section 2 gives some key facts on the theory behind Usain Boltz. Section 3 describes our running example. Section 4 shows how to use the library and explain some relevant features. Then Section 5 presents a performance comparison with other existing tools. And finally Section 6 concludes with possible improvements.

2 The Boltzmann method in a nutshell

The Boltzmann method works on *combinatorial specifications*, which are at the core of analytic combinatorics [12]. They offer a language with basic constructions and combinators allowing to describe data structures by a system of equations. The most important such constructions are the disjoint union, the Cartesian product, and some base cases, which we illustrate here in an example. A more complete list can be found in [12, Theorem I.1 p.27]. Here is a specification of the set of binary trees in this formalism:

$$\mathcal{B} = \mathcal{E} + \mathcal{Z} \times \mathcal{B} \times \mathcal{B}. \tag{1}$$

This reads as follows: a binary tree $\mathcal{B}$ is either a leaf $\mathcal{E}$ or an internal node with two children $\mathcal{Z} \times \mathcal{B} \times \mathcal{B}$. The $\times$ symbol represents a Cartesian product (you need two $\mathcal{B}$s to form a pair of children) and the $+$ symbol represents a *disjoint* union.

A key component of such descriptions is the notion of *size*. Here the symbol $\mathcal{Z}$ encodes the fact that an internal node "accounts for one" in the size of a tree, hereby defining the size of a binary tree as its number of internal nodes. By opposition, the symbol $\mathcal{E}$, represents a single object of size zero. It is essential for the theory of Boltzmann samplers that the number of objects of a given size described by such a specification remains finite. This gives a well-founded notion of "uniform object of size n". Given a set $\mathcal{A}$ described by such a specification, the Boltzmann method provides an automatic way to build algorithms for the random sampling of elements of $\mathcal{A}$ according to a specific probability distribution: the *Boltzmann model* [11, p. 581]. It offers two guaranties. First, two objects of the same size have the same probability to be drawn (uniformity by size). And second, when coupled with (early) rejection as described in [11, p. 601], it allows

[3] available at https://gitlab.com/ParComb/usain-boltz

to draw objects of size close to a target size n (that is in $[(1-\epsilon)n; (1+\epsilon)n]$ for some $\epsilon > 0$) in *linear time*.

3 Example of application: RNA secondary structures

As an illustrative example that we will follow along this paper, consider the RNA (RiboNucleic Acid) secondary structures as defined in [22]. These are a simplified view of the structures resulting from the natural folding of RNA strands in their environment. According to [22], RNA strands (chains of nucleotides A, U, C, and G) fold in a hierarchical fashion where matching pairs ($A-U$ and $C-G$) create non-crossing links (chemical bounds), as modelled below:

$$\begin{aligned} \mathcal{S} &= \mathcal{B} \times \mathcal{Z} \times (\mathcal{S} + \mathcal{E}) + \mathcal{B} \times \mathcal{Z} \times \mathcal{S} \times \mathcal{Z} \times (\mathcal{S} + \mathcal{E}) \\ \mathcal{B} &= \mathcal{U}_A + \mathcal{U}_U + \mathcal{U}_C + \mathcal{U}_G \end{aligned} \tag{2}$$

Here, $\mathcal{S}$ represents a secondary structure, $\mathcal{Z}$ counts nucleotides, $\mathcal{E}$ represents an empty strand, and $\mathcal{B}$ distinguishes between the four kinds of nucleotides using *marker* variables. The marker construction $\mathcal{U}_x$ does not contribute to the size (like $\mathcal{E}$), but it "marks" and keeps a count of some features of the data structure. This allows to get statistics from the sampler and to bias the generation (see below). The second term $\mathcal{B} \times \mathcal{Z} \times \mathcal{S} \times \mathcal{Z} \times (\mathcal{S} + \mathcal{E})$, encodes a link between the first nucleotide $\mathcal{B} \times \mathcal{Z}$ and a matching nucleotide $\mathcal{Z}$, within a structure.

4 Usain Boltz

The first step to get a Boltzmann sampler using Usain Boltz is to give it a combinatorial specification. They are represented in Python using a hierarchy of classes, one for each possible construction and one to represent the whole system of equations. The most notable constructions are illustrated below on the example of RNA secondary structures and the complete list can be found online in our package's documentation[4] .

```
B, S, z, empty = RuleName("B"), RuleName("S"), Atom(), Epsilon()
A, U, C, G = Marker("A"), Marker("U"), Marker("C"), Marker("G")
grammar = Grammar({S: B * z * (S + empty) + B * z * S * z * (S + empty),
                   B: A + U + C + G})
```

Note that to be able to bind $\mathcal{B}$ and $\mathcal{S}$ in the `Grammar`, we must declare `S` and `B` as `RuleNames` beforehand. Also, since specifications are regular Python expressions it is possible to define expressions outside of the grammar as short-hands, as we did here with `z` and `empty` for instance.

Then, one needs to create a `Generator` instance, which initialises the `C++` sampler that runs under the hood, and provides an interface to it. The simplest way to call this sampler is then via the `sample` method which takes a size window as an argument and outputs a `result` object storing a generated structure (of size in the window) in `result.obj` and some statistics in `result.sizes`.

[4] https://usain-boltz.readthedocs.io/en/latest/generated/usainboltz.grammar.html

```
generator = Generator(grammar)
result = generator.sample((10000, 15000))
# result.sizes = {A: 2080, U: 2063, G: 2087, C: 2110, z: 10558}
```

It is also possible to *tune* the generator (thanks to paganini [4]) towards producing more `A`s and to apply size-based rejection to `A`, to bias its output. For instance you can tell that you want 3000 Adenine nucleotides in expectation and reject any quantity lower than 2800 as so:

```
biased_gen = Generator(grammar, expectations={A: 3000, z: 10000})
res = biased_gen.sample({z: (10000, 15000), A: (2800, 15000)})
# res.sizes = {U: 2788, C: 2760, G: 2700, A: 2803, z: 13986}
```

4.1 Generator optimisations

Simulation Although the generation of objects of size in $[(1-\epsilon)n; (1+\epsilon)n]$ is linear in average, it has a non-negligible cost in practice because of the useless allocations incurred by the rejections. To circumvent this issue, we propose an alternative rejection procedure that "simulates" the generation of an object by only computing its size rather than actually constructing it. When a size in the targeted size window is found, we reset the pseudo random number generator to the state in which it was just before the simulation and the actual generator is run instead of the simulator. This technique gives a significant speed-up as shown in Figure 1.

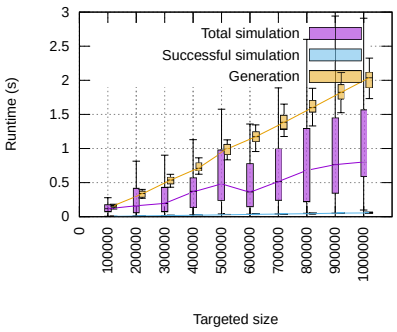

Fig. 1. Comparison of the runtime spent in the simulation and generation algorithms for RNA secondary structures (2) where $\epsilon = 0.1$ and n varies. In pink is the total simulation time, in blue is the portion of that runtime spent in the last simulation (the successful one), and in yellow is the runtime spent in the final generation. For selected values of n, we CPU-timed the algorithms 51 times with different seeds and show here their median and interquartile range (IQR) in a box plot. The whiskers extend to the most distant measure that lies within 1.5 times the IQR.

Non-recursive implementation Another aspect of our implementation is that we implemented both the simulation and the generation routines in a non-recursive fashion. The original description of the algorithm is not tail-recursive and uses a stack space that is of the same order as the *height* of the generated tree, that is $\sqrt{n}$ is most cases (see [14]). This cannot be neglected since stack space is often a limited resource. Our implementation takes the form of a stack machine described in the PhD thesis of one of the authors [25, p. 120].

4.2 Builders

A last feature of our implementation, which we believe is unique to Usain Boltz and makes it easy to integrate inside another system, is the "builder" mechanism. The idea is to let to the user of the library the choice of how the generated objects are constructed rather than enforcing one data representation. This is achieved by letting the user pass to the generator a "builder" function for each symbol of the specification. These functions take a partially built structure as an argument (a nested tuple in which all recursive sub-structures are already fully built) and complete the construction.

For instance, here is an RNA secondary structure of size 15 generated with the default builders using tuples to represents the elements of a Cartesian product:

```
(U, 'z', (C, 'z', 'epsilon'), 'z', (U, 'z', (A, 'z', (G, 'z', (G, 'z',
(C, 'z', 'epsilon')))), 'z', (C, 'z', (G, 'z', (C, 'z', (A, 'z', (U, 'z',
'epsilon'), 'z', 'epsilon'))))))
```

One possible, more suitable, data representation is to use list of markers to store the list of bases, and to store an offset together with the first base of each link. To use this representation, we must pass a builder for `S` to the generator. This builder will receive either a tuple of type `B * z * (S + empty)` or a tuple of type `B * z * S * z * (S + empty)` where the `S` objects have *already* been built. Note that we keep the default builder for `B` as it already does what we want.

```
dual = {A: U, U: A, C: G, G: C}
build_S_or_E = union_builder(lambda x: x, lambda eps: [])
def build_prefix(t):  # case: B * z * (S + Epsilon())
    base, z, se = t
    return [base] + build_S_or_E(se)
def build_matching(t):  # case: B * S * z * (S + Epsilon())
    base, z, s, z, se = t
    return [(base, len(s) + 1)] + s + [dual[base]] + build_S_or_E(se)
generator.set_builder(S, union_builder(build_prefix, build_matching))
```

Note the use of the `union_builder` function here. In the case of the union of two classes (*e.g.* `(S + empty)`), in order to avoid potential ambiguities, the generator tags the generated object with an integer indicating from which component of the union it comes from. The `union_builder` function automates the process of calling the right builder depending on this tag, so that the user only has to specify a sub-builder for each component of the union. For instance, to build an element from `(S + empty)`, we must provide two builders. In case we receive an `S`, the object has already been built so we pass the identity function, and in case we receive an `Epsilon()`, we return the empty secondary structure `[]`. Finally we pass the builder to the generator by calling the `set_builder` method. Using the same seed as in the previous example the generator now yields:

```
[(U, 2), C, A, (U, 5), A, G, G, C, A, C, G, C, (A, 2), U, U]
```

5 Performance comparison

At the moment, few tools currently implement the Boltzmann method. An OCaml prototype called Arbogen [10] has been developed by various people including the authors. It has few features but implements the simulation mechanism. Another tool called Boltzmann brain has been written in Haskell [1,4,3], in the form of a compiler that takes a Haskell data type as an input and produces a Haskell sampler, and also as a template Haskell library. Unfortunately, due to some (probably minor) bugs in Boltzmann brain, we could not compare its performance with that of Usain Boltz on the RNA example. So we compared the three tools on the simpler example of binary trees (1) which works across all tools. The results are presented in Figure 2 and the sources of the experiments are available at https://gitlab.com/ParComb/usain-boltz/-/tree/master/benchmarks.

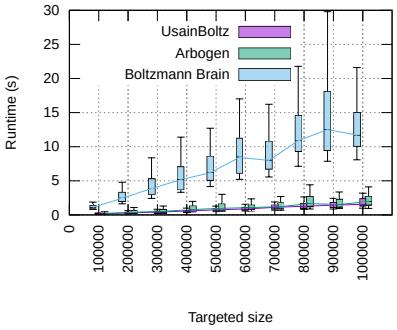

Fig. 2. Comparison of the runtimes of Usain Boltz, Arbogen, and Boltzmann brain for generating binary trees. The setup is the same as for the simulation benchmark of Figure 1: for each size, run the generator 51 times and draw some box plots to describe the runtime discrepancy between the different runs. We can see a clear difference here between Boltzmann brain and the other two tools, which we explain by the simulation trick (see Section 4.1).

6 Perspectives

Usain Boltz has more features than we could present in this paper, but the literature on Boltzmann sampling is rich of extensions we want to implement. For instance, the so-called Pólya operators, allowing to deal with symmetries, have been added to the Boltzmann framework in [13,6]. We already implemented the most common one (MSet) but some work remains to be done in this direction. Moreover, Usain Boltz also supports labelled specifications [11, Sec.4, p. 593], but no form of constrained labellings [5,9] yet, which is also planned for the future. Finally, some internal tuning of the algorithms is delegated to the open source paganini [4,2] library. But in rare occasions we fall out of its scope and need a more general tool such as NewtonGF [21]. However it is bound to the proprietary Maple computer algebra system which makes the interface difficult. We would like to provide our own Python implementation of this tool.

References

1. Bendkowski, M.: boltzmann-brain: Analytic sampler compiler for combinatorial systems, https://github.com/maciej-bendkowski/boltzmann-brain
2. Bendkowski, M.: paganini: Multiparametric tuner for combinatorial specifications, https://github.com/maciej-bendkowski/paganini
3. Bendkowski, M.: Automatic compile-time synthesis of entropy-optimal boltzmann samplers (2022)
4. Bendkowski, M., Bodini, O., Dovgal, S.: Polynomial tuning of multiparametric combinatorial samplers. In: 2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO). pp. 92–106. SIAM (2018)
5. Bodini, O., Roussel, O., Soria, M.: Boltzmann samplers for first-order differential specifications. Discrete Applied Mathematics **160**(18), 2563–2572 (Dec 2012). https://doi.org/10.1016/j.dam.2012.05.022
6. Bodirsky, M., Fusy, É., Kang, M., Vigerske, S.: Boltzmann samplers, pólya theory, and cycle pointing. SIAM Journal on Computing **40**(3), 721–769 (2011)
7. Canou, B., Darrasse, A.: Fast and sound random generation for automated testing and benchmarking in objective caml. In: Proceedings of the 2009 ACM SIGPLAN workshop on ML. pp. 61–70 (2009)
8. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. Acm sigplan notices **46**(4), 53–64 (2011)
9. Dien, M.: Processus concurrents et combinatoire des structures croissantes : analyse quantitative et algorithmes de génération aléatoire. Ph.D. thesis, Université Pierre et Marie Curie - Paris VI (9 2017), http://www.theses.fr/2017PA066210
10. Dien, M., Genitrini, A., Ghanem, M., Pépin, M., Peschanski, F., Zhan, X.: arbogen: a fast uniform random tree generator, https://github.com/fredokun/arbogen
11. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Boltzmann Samplers for the Random Generation of Combinatorial Structures. Combinatorics, Probability and Computing **13**(4-5), 577–625 (Jul 2004)
12. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge Univ. Press (2009)
13. Flajolet, P., Fusy, É., Pivoteau, C.: Boltzmann sampling of unlabelled structures. In: 2007 Proceedings of the Fourth Workshop on Analytic Algorithmics and Combinatorics (ANALCO). pp. 201–211. SIAM (2007)
14. Flajolet, P., Odlyzko, A.M.: The average height of binary trees and other simple trees. Journal of Computer and System Sciences **25**, 171–213 (jan 1981)
15. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of labelled combinatorial structures. Theor. Comput. Sci. **132**(2), 1–35 (1994). https://doi.org/10.1016/0304-3975(94)90226-7, https://doi.org/10.1016/0304-3975(94)90226-7
16. Jockusch, W., Propp, J., Shor, P.: Random domino tilings and the arctic circle theorem (1998)
17. Levochik: A lozenge tiling of a hexagon chosen uniformly at random, with the "frozen" tiles being depicted in white. (arctic circle theorem.) (2012), https://en.wikipedia.org/wiki/Aztec_diamond#/media/File:Arctic_Circle.svg
18. Martínez, C., Molinero, X.: A generic approach for the unranking of labeled combinatorial classes. Random Structures & Algorithms **19**(3-4), 472–497 (2001)
19. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) Model Driven Architecture - Foundations and Applications. pp. 130–145. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

20. Nijenhuis, A., Wilf, H.: Combinatorial Algorithms: For Computers and Hard Calculators. Academic Press, Inc., USA, 2nd edn. (1978)
21. Pivoteau, Carine and Salvy, Bruno and Soria, Michele: Algorithms for combinatorial structures: Well-founded systems and Newton iterations. Journal of Combinatorial Theory, Series A **119**(8), 1711–1773 (Nov 2012). https://doi.org/10.1016/j.jcta.2012.05.007
22. Ponty, Y.: Ensemble algorithms and analytic combinatorics in rna bioinformatics and beyond (2020)
23. Propp, J.G., Wilson, D.B.: Exact sampling with coupled markov chains and applications to statistical mechanics. Random Structures & Algorithms **9**(1-2), 223–252 (1996). https://doi.org/10.1002/(SICI)1098-2418(199608/09)9:1/2<223::AID-RSA14>3.0.CO;2-O
24. Python Core Team: Python, https://www.python.org
25. Pépin, M.: Quantitative and algorithmic analysis of concurrent programs. Ph.D. thesis, Sorbonne Université (9 2021)
26. The Sage Developers: SageMath, the Sage Mathematics Software System (Version 8.5.0) (2018), `https://www.sagemath.org`