A Standalone Python Library for Enumeration of Combinatorial Structures

Sam Pritt (BSE 2017)

UNDER THE ADVISEMENT OF JÉRÉMIE LUMBROSO

1 INTRODUCTION

Combinatorial enumeration is the process of counting objects of finite size using analytic methods. For instance, there are 14 different configurations of the binary trees of size 5 if we define the size of a binary tree to be the number of its internal nodes. It is often possible to enumerate such objects through various algebraic methods such as Lagrange inversions and Taylor series expansions, and an entire field of research has been devoted to such methods [3]. In this work we take an algorithmic approach with the aim of developing an automated software tool for enumeration of arbitrarily complex combinatorial objects.

1.1 MOTIVATION

This work was motivated by a colleague's research involving enumeration of complex combinatorial structures [1]. Bahrani and Lumbroso were using Maple's analytic combinatorial library `combstruct` (previously Gaia) [11] in the Maple software but were unable to enumerate certain structures due to the limited functionality of the library. Additionally, because `combstruct` is an obscure proprietary language there was no way to extend the library or integrate it with another platform to support these additional stuctures.

Although several tools exist, they are each limited in one way or another. In addition to its limited functionality and lack of compatability with other platforms, `combstruct` was developed in the 1990s and is thus far from current. MuPAD, a proprietary MatLab package with some ability to do combinatorial computations, was discontinued in 2008 [7]. Since 2005 there have been talks to port the `combstruct` library to Sage, an open source algebraic tool, but over a decade later not even a prototype has been released [2].

With the only available tools being clunky proprietary libraries developed in the previous decade or earlier, we can safely conclude that there is need for an up to date, easily accessible and user friendly tool for combinatorial enumeration.

1.2 RELATED WORK

Unfortunately the literature on this subject is quite sparse. One of the main sources of information for this work was Zimmermann's 1991 dissertation [10], which lays out many of the fundamental algorithms for combinatorial enumeration. It's a rather confusing paper written entirely in French, and a decent amount of email correspondence was necessary to obtain clarification on various parts of the paper. However, many of the algorithms underlying the valuation of rules and generation of coefficients discussed in Section 4 were presented by Zimmermann in this paper.

Flajolet, Zimmermann and Van Cutsem released a paper in 1994 on random generation of labeled combinatorial structures [5] which was also useful for this work. A companion paper for unlabeled structures was written by the same authors but never published [4]. Although these papers are not aimed at enumeration, the algorithms for converting symbolic specifi-

cations to Chomsky Normal Form as discussed in Section 3 are fundamental to this work.

Other sources for this work did not relate directly to algorithmic enumeration but served as helpful references. In particular, Flajolet and Sedgewick's book on analytic combinatorics [3] and Lumbroso and Morcrette's guide [6] provided much of the theory behind combinatorial classes, constructions and generating functions.

1.3 Goals

The goal of this work was to develop an algorithmic enumeration tool to spare researchers the headaches associated with `combstruct` and similar tools. The tool would be open source, standalone, extendible, and legibly coded in a well known language such as Python or C. Additionally, we would clearly document all of the algorithms in a paper so that for the first time a single, complete reference on algorithmic enumeration is available to researchers.

2 Symbolic Equations

2.1 Basic Definitions

The theory of analytic combinatorics was developed in recent years largely by Flajolet and Sedgewick [3]. An important component of the theory is the use of *symbolic equations* to generate the enumerations of combinatorial objects. An example of a symbolic equation, or *symbolic specification*, is the equation for binary trees:

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

In this case we have defined a binary tree recursively as either a leaf or a pair of binary trees. According to the *symbolic method*, valid combinatorial classes - including the union, product, sequence, set and cycle classes studied in detail in this work - can be reduced directly to algebraic generating functions in order to determine their *counting sequences*.

Definition 1. A *combinatorial class* $\mathcal{A}$ is a finite or denumerable set on which is defined a *size* function such that for every size there is only a finite number of elements. [6]

Definition 2. Let $\mathcal{A}$ be an *unlabeled* combinatorial class, and let a_n be its counting sequence. We define the *ordinary generating function* $A(z)$ of the class $\mathcal{A}$ as follows [6]:

$$A(z) = \sum_{n=0}^{\infty} a_n z^n$$

The atom, union, product and sequence classes are constructed from the definition of the generating function. The formal proofs of these constructions are beyond the scope of this work [6].

- *Atom.* The atomic class $\mathcal{Z}$ contains a single element of size 1 and has generating function $Z(z) = z$. The class $\mathcal{Z}^N$ represents an atom of size N. The atom of size 0 is referred to as the neutral class, denoted ε and has generating function $E(z) = 1$.

- *Union.* The union class is the combinatorial sum of two disjoint classes. Thus the generating function is $A(z) = B(z) + C(z)$
- *Product.* The product class forms all possible ordered pairs of two classes. The generating function is the Cartesian product: $A(z) = B(z) \cdot C(z)$
- *Sequence.* The sequence class is defined by the infinite sum $\textsc{Seq}(\mathcal{B}) = \varepsilon + \mathcal{B} + (\mathcal{B} \times \mathcal{B}) + (\mathcal{B} \times \mathcal{B} \times \mathcal{B}) + \ldots$ The generating function is the sum of the geometric series: $A(z) = \frac{1}{1-B(z)}$

Definition 3. Let $\mathcal{A}$ be a *labeled* combinatorial class, and let a_n be its counting sequence. We define the *exponential generating function* $A(z)$ of the class $\mathcal{A}$ as follows [6]:

$$A(z) = \sum_{n=0}^{\infty} a_n \frac{z^n}{n!}$$

Note that we can rearrange the above to extract the coefficients of a combinatorial class from its generating function: $a_n = n![z^n]A(z)$. This process will be studied in Section 4. We use exponential generating functions to construct classes of exponential size such as permutations. The constructions of the labeled union, product and sequence classes are identical to the corresponding unlabeled constructions.

- *Set.* Consider that sets are permutations of sequences, which implies a cardinality ratio of $\frac{|\textsc{Set}_k(\mathcal{B})|}{|\textsc{Seq}_k(\mathcal{B})|} = 1/k!$ (there are $k!$ permutations of a size k sequence). The resulting generating function is $A(z) = \sum_{k\geq 0} \frac{1}{k!} B(z)^k$ which is the formal power series expansion for the exponential function: $A(z) = e^{B(z)}$
- *Cycle.* Similarly, consider that cycles are cyclic permutations of sequences which implies a cardinality ratio of $\frac{|\textsc{Cyc}_k(\mathcal{B})|}{|\textsc{Seq}_k(\mathcal{B})|} = 1/k$ (there are k cyclic permutations of a size k sequence). The resulting generating function is $A(z) = \sum_{k\geq 0} \frac{1}{k} B(z)^k$ which simplifies to $A(z) = \log(\frac{1}{1-B(z)})$

The constructed generating functions for labeled classes are summarized in **Figure 1**.

Structure	*EGF*
$\{\varepsilon\}$	1
$\{\mathcal{Z}\}$	z
$\mathcal{A} + \mathcal{B}$	$A(z) + B(z)$
$\mathcal{A} \times \mathcal{B}$	$A(z) \cdot B(z)$
$\textsc{Seq}(\mathcal{A})$	$1/(1 - A(z))$
$\textsc{Set}(\mathcal{A})$	$e^{A(z)}$
$\textsc{Cyc}(\mathcal{A})$	$\log \frac{1}{1-A(z)}$

Figure 1. Labeled combinatorial classes and their generating functions.

2.2 UNLABELED CONSTRUCTIONS

It is important to distinguish between the labeled and unlabeled contexts. In a *labeled* object, every atom is assumed to be unique and distinguishable from the rest of the structure. In an *unlabeled* object, two or more subparts may be identical. For union, product and sequence constructions, a labeled object is essentially a permutation of the corresponding unlabeled object. There are thus $n!$ as many labeled as unlabeled objects of size n and the generating functions are otherwise identical. For set and cycle constructions the difference between labeled and unlabeled is more involved. For instance, an unlabeled multiset can be sorted so that its identical elements are grouped together, resulting in a product of sequences. From this point forward all references to unlabeled sets are in fact references to multisets, or sets where repeated elements are permissible (ordinary sets do not allow repeated elements, and thus do not exist as unlabeled objects). For simplicity we do not consider powersets in this work. Enumeration of unlabeled multisets and cycles relies partially on *Pólya theory*, a discussion of which is beyond the scope of this work. The formal proofs for these definitions are given in [3].

- *Unlabeled set.* If we sort the elements of the set so that identical elements occur together, we see that the set is equivalent to a product over sequences of identical elements:

  $$\textsc{Set}(\mathcal{B}) = \prod_{\beta \in \mathcal{B}} \textsc{Seq}\left(\{\beta\}\right)$$

  Substitution of the product and sequence constructions and simple algebra yields the generating function $A(z) = e^{C(z)}, C(z) = \sum_{k=1}^{\infty} \frac{1}{k} B(z^k)$

- *Unlabeled cycle.* The generating function for cycles is derived from multivariate generating functions which are beyond the scope of this work [3]: $A(z) = \sum_{k=1}^{\infty} \frac{\varphi(k)}{k} \log \frac{1}{1-B(z^k)}$ where φ is Euler's totient function.

The constructed generating functions for unlabeled classes are summarized in **Figure 2**.

Structure	*OGF*
$\{\varepsilon\}$	1
$\{\mathcal{Z}\}$	z
$\mathcal{A} + \mathcal{B}$	$A(z) + B(z)$
$\mathcal{A} \times \mathcal{B}$	$A(z) \cdot B(z)$
$\textsc{Seq}(\mathcal{A})$	$1/(1 - A(z))$
$\textsc{Set}(\mathcal{A})$	$\exp(\sum_{k=1}^{\infty} \frac{1}{k} A(z^k))$
$\textsc{Cyc}(\mathcal{A})$	$\sum_{k=1}^{\infty} \frac{\varphi(k)}{k} \log \frac{1}{1-A(z^k)}$

Figure 2. Unlabeled combinatorial classes and their generating functions.

2.3 CARDINALITY RESTRICTIONS

Often we want to restrict the number of components in sequences, sets and cycles to a fixed cardinality k. For example, $\textsc{Seq}_k(\mathcal{A})$, $\textsc{Seq}_{\geqslant k}(\mathcal{A})$ and $\textsc{Seq}_{\leqslant k}(\mathcal{A})$ denote the sequences with exactly k elements, greater than or equal to k elements, and less than or equal to k elements respectively.

- $A = \textsc{Seq}_k(\mathcal{B})$. Observe that $\textsc{Seq}_k(\mathcal{B}) = B^k$. This is just a single term of the infinite expansion for the unrestricted sequence (alternatively, consider that a sequence of elements restricted to size k is just k elements). The corresponding generating function is $A(z) = B^k(z)$
- $A = \textsc{Set}_k(\mathcal{B})$. Recall from above the relationship between labeled sets and sequences: $\frac{|\textsc{Set}_k(\mathcal{B})|}{|\textsc{Seq}_k(\mathcal{B})|} = 1/k!$ Thus the generating function is $A(z) = B^k(z)/k!$
- $A = \textsc{Cyc}_k(\mathcal{B})$. Again, recall from above the relationship between labeled cycles and sequences: $\frac{|\textsc{Cyc}_k(\mathcal{B})|}{|\textsc{Seq}_k(\mathcal{B})|} = 1/k$. Thus the generating function is $A(z) = B^k(z)/k$

We can further derive generating functions for sequences, sets and cycles with $\leq$ and $\geq$ restrictions by summation. For example, $A = \textsc{Seq}_{\leqslant k}(\mathcal{B})$ has the generating function $A(z) = \sum_{j=0}^{k} B^j(z)$ and $A = \textsc{Seq}_{\geqslant k}(\mathcal{B})$ has the generating function $A(z) = 1/(1-B(z)) - \sum_{j=0}^{k-1} B^j(z)$. Restricted sets and cycles are derived similarly. For simplicity we do not include these generating functions here as they are not essential for the enumeration algorithm.

The constructed generating functions for labeled classes with cardinality restrictions are summarized in **Figure 3**.

Structure	*EGF*
$\textsc{Seq}_k(\mathcal{A})$	$A^k(z)$
$\textsc{Set}_k(\mathcal{A})$	$A^k(z)/k!$
$\textsc{Cyc}_k(\mathcal{A})$	$A^k(z)/k$

Figure 3. Labeled restricted combinatorial classes and their generating functions.

For unlabeled sets and cycles with cardinality restrictions we rely on bivariate generating functions [3].

- $A = \textsc{Set}_k(\mathcal{B})$. The unlabeled sets with cardinality restrictions are given by the bivariate generating function:

$$A(z,u) = \exp\Big(\sum_{\ell=1}^{\infty} \frac{u^\ell}{\ell} B(z^\ell)\Big)$$

from which we can extract the sets of size k:

$$A(z) = [u^k]\exp\Big(\sum_{\ell=1}^{\infty} \frac{u^\ell}{\ell} B(z^\ell)\Big)$$

- $A = \textsc{Cyc}_k(\mathcal{B})$. Similarly:

$$A(z) = [u^k]\exp(\sum_{\ell=1}^{\infty} \frac{\varphi(\ell)}{\ell} \log \frac{1}{1-u^\ell B(z^\ell)})$$

Once again for simplicity we do not include the generating functions for $\leq$ and $\geq$ restrictions. These are discussed in greater detail in the following section.

The constructed generating functions for unlabeled classes with cardinality restrictions are summarized in **Figure 4**.

Structure	*OGF*
$\textsc{Seq}_k(\mathcal{A})$	$A^k(z)$
$\textsc{Set}_k(\mathcal{A})$	$[u^k]\exp(\sum_{\ell=1}^{\infty} \frac{u^\ell}{\ell} A(z^\ell))$
$\textsc{Cyc}_k(\mathcal{A})$	$[u^k]\exp(\sum_{\ell=1}^{\infty} \frac{\varphi(\ell)}{\ell} \log \frac{1}{1-u^\ell A(z^\ell)})$

Figure 4. Unlabeled restricted combinatorial classes and their generating functions.

3 Standard Specifications

We say a symbolic equation is *decomposable* if it can be reduced to Chomsky Normal Form, also referred to as a *standard specification* or *standard form* [5]. A standard specification is a system of equations which satisfies the original symbolic equation where each equation contains only a single binary operator or an atom. Throughout this paper we refer to equations in Chomsky Normal Form as *rules*. For example, the symbolic specification for binary trees

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

reduces by simple binary substitution to the following rules:

$$\mathcal{B} = \mathcal{U} + \mathcal{V}, \mathcal{U} = \mathcal{Z}, \mathcal{V} = \mathcal{B} \cdot \mathcal{B}$$

The simple binary conversion is due to the polynomial generating functions for unions, products and atoms. Conceptually, the algorithm can be represented by a binary tree where internal nodes are the union and product operators and leaves are the terminal objects (atoms) and the recursive substructures. In practice, the implementation is roughly polynomial in the size of the standard form.

3.1 Pointing

Some classes are more complex and cannot be converted by simple substitution. It is nontrivial to reduce the specifications for sets and cycles to standard form because these classes have non-polynomial generating functions as discussed in the previous section and shown in **Figure 1**. We reduce complex symbolic equations to standard form using differential substitutions which are referred to in the literature as "pointing" or "marking" [5] [6]. Taking the exponential generating function for labeled sets,

$$A(z) = e^{B(z)}$$

we differentiate both sides:

$$A'(z) = B'(z)e^{B(z)}$$

Combinatorically, we rewrite the derivative using the differential operator Θ, which we call theta notation:

$$\Theta\mathcal{A} = \mathcal{A} \cdot \Theta\mathcal{B}$$

Symbolically, this is equivalent to the "pointing" operation, in which we count all different ways of selecting a node of the tree, as illustrated in **Figure 5**. The consequence is that each tree of the original class $\mathcal{B}$ will be duplicated however many ways there are to distinguish a node, in other words $C = \Theta A$ implies the enumeration $c_k = k \cdot a_k$ and we have reduced the non-polynomial generating function to a polynomial rule in Chomsky Normal Form (that is, a single binary operator excluding the Θ operator). Thus the class of Cayley trees defined by the symbolic equation

$$\mathcal{H} = \mathcal{Z} \times \textsc{Set}(\mathcal{H})$$

reduces to the following standard specification:

$$\mathcal{H} = \mathcal{Z} \cdot \mathcal{V}, \mathcal{V} = \textsc{Set}(\mathcal{H}), \Theta\mathcal{V} = \mathcal{V} \cdot \Theta\mathcal{H}$$

Note that we have included the rule $\mathcal{V} = \textsc{Set}(\mathcal{H})$ in the standard specification even though it is not in proper Chomsky Normal Form. This was determined through our investigation and correspondence with Zimmermann to be necessary to prevent cyclic dependencies in the evaluation of coefficients as discussed in more detail below.

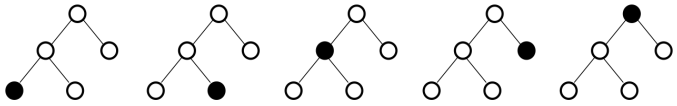

Figure 5. There are k ways to point an object of size k.

3.2 Diagonalization

The conversion to standard form becomes even more complex for unlabeled sets and cycles due to the summation of diagonal terms like $A(z^k)$ representing repeated elements. The *diagonal* operator [3] [4] replaces such terms and is defined as follows:

$$\Delta^{(k)} A(z) = A(z^k)$$

Intuitively, the diagonal represents all k-length sequences of identical elements of A. We also define a *generalized diagonal* [4] as shown:

$$\Delta_{\{u(k)\}} A(z) = \sum_{k=1}^{\infty} u(k) A(z^k)$$

Here $u(k)$ can receive any function. Consider the case of unlabeled sets, $\mathcal{A} = \textsc{Set}(\mathcal{B})$:

$$A(z) = e^{C(z)},\ C(z) = \sum_{k=1}^{\infty} \tfrac{1}{k} B(z^k)$$

We replace $C(z)$ in the above expression with $\Delta_{\{1/k\}} B(z)$:

$$A(z) = e^{\Delta_{\{1/k\}} B(z)}$$

and the pointing operator is used to reduce the equation to standard form.

$$\Theta A(z) = A(z) \cdot \Delta_{\{1\}} \Theta B(z)$$

Once again we have reduced a non-polynomial generating function to a polynomial rule in Chomsky Normal Form. The class of rooted unlabeled trees defined by the equation

$$\mathcal{G} = \mathcal{Z} \times \textsc{Set}(\mathcal{G})$$

reduces to the following standard specification:

$$\mathcal{G} = \mathcal{A} \cdot \mathcal{B},\ \mathcal{A} = \mathcal{Z},\ \mathcal{B} = \textsc{Set}(\mathcal{G}),\ \Theta\mathcal{B} = \mathcal{B} \cdot \mathcal{C},\ \mathcal{C} = \Delta_{\{1\}} \mathcal{D},\ \mathcal{D} = \Theta\mathcal{G}$$

3.3 Conversion

Every decomposable structure is convertible to an equivalent standard specification. Flajolet, Zimmermann and Van Cutsem give a conversion algorithm for labeled symbolic equations composed of atoms, unions, products, sequences, sets and cycles [5]. Additionally, conversion rules for unlabeled sets and cycles are given in the unpublished companion paper [4]. The conversion rules are derived from the corresponding generating functions by substitution, pointing and diagonalization as described in the earlier parts of this section. The conversion algorithm which we present here is based on the work of Flajolet, Zimmermann and Van Cutsem as well as on our own study of the theory and conclusions drawn from implementation.

- **Polynomial.** As mentioned above, polynomials (unions, products and atoms) convert by simple binary substitutions such as the following: $\mathcal{A} = \mathcal{B} + \mathcal{C} \Rightarrow \mathcal{A} = \mathcal{U} + \mathcal{V}, \mathcal{U} = \mathcal{A}, \mathcal{V} = \mathcal{B}$. Thus each substitution results in at least one new sub-rule which must be recursively converted. We found that in order to avoid redundancies, each new sub-rule should be looked up in the partially complete standard form in case it has already been converted. For example, if the new sub-rule is $\mathcal{W} = \mathcal{Z} + \mathcal{X}$ and the standard form already includes a rule $\mathcal{Y} = \mathcal{Z} + \mathcal{X}$ then it would be incorrect to add a rule for $\mathcal{W}$ since $\mathcal{W} = \mathcal{Y}$. This sort of redundancy can be prevented by a linear traversal through the standard form.

- **Sequence.** The conversion follows algebraically from the generating function for sequences, $A(z) = \frac{1}{1-B(z)}$:

$$\mathcal{A} = \textsc{Seq}(\mathcal{B}) \rightarrow \mathcal{A} = 1 + \mathcal{A} \cdot \mathcal{B}$$

  We convert sequences with cardinality restrictions similarly: $\textsc{Seq}_k(\mathcal{B})$ converts to B^k, $\textsc{Seq}_{\leqslant k}(\mathcal{B})$ converts to $1 + B + B^2 + ... + B^k$, and $\textsc{Seq}_{\geqslant k}(\mathcal{B})$ converts to $B^k \cdot \textsc{Seq}(\mathcal{A})$

- **Set.** The conversion is achieved by pointing:

$$\mathcal{A} = \textsc{Set}(\mathcal{B}) \rightarrow \Theta\mathcal{A} = \mathcal{A} \cdot \Theta\mathcal{B}$$

  A pointed set of k elements is equivalent to the Cartesian product of a single pointed element and the remainder of the set, which is a set of size $k-1$. Thus sets with cardinality restrictions convert by recurrence:

$$\mathcal{A} = \textsc{Set}_k(\mathcal{B}) \rightarrow \Theta\mathcal{A} = \Theta\mathcal{B} \cdot \textsc{Set}_{k-1}\mathcal{B}$$

  The same recurrence applies to $\mathcal{A} = \textsc{Set}_{\geqslant k}(\mathcal{B})$ and $\mathcal{A} = \textsc{Set}_{\leqslant k}(\mathcal{B})$. The base case for $\textsc{Set}_k(\mathcal{B})$ is $\textsc{Set}_1(\mathcal{B})$ which converts to B because a size 1 set of elements is just a single element. The base case for $\textsc{Set}_{\leqslant k}(\mathcal{B})$ is $\textsc{Set}_{\leqslant 1}(\mathcal{B})$ which converts to $1 + B$ because it is either a single element or the neutral class (i.e. a set of cardinality 0). The base case for $\textsc{Set}_{\geqslant k}(\mathcal{B})$ is just the unrestricted set, $\textsc{Set}(\mathcal{B})$, for which we have already discussed the conversion.

- **Cycle.** A pointed cycle is equivalent to a single pointed element and the remainder of the cycle. By definition a cycle with an element missing is a sequence, $\textsc{Seq}(\mathcal{B})$. This gives us the conversion:

$$\mathcal{A} = \textsc{Cyc}(\mathcal{B}) \rightarrow \Theta\mathcal{A} = \Theta\mathcal{B} \cdot \textsc{Seq}(B)$$

  In the case of cardinality restrictions, a pointed cycle of size k reduces to a pointed element and a sequence of size $k-1$, so unlike sets, cycles with cardinality restrictions do not convert by recurrence.:

$$\mathcal{A} = \textsc{Cyc}_k(\mathcal{B}) \rightarrow \Theta\mathcal{A} = \Theta\mathcal{B} \cdot \textsc{Seq}_{k-1}\mathcal{B}$$

  The same conversion applies to $\mathcal{A} = \textsc{Cyc}_{\geqslant k}(\mathcal{B})$ and $\mathcal{A} = \textsc{Cyc}_{\leqslant k}(\mathcal{B})$.

Unlabeled unions, products and sequences have the same generating functions and thus convert to standard form by the same rules. Unlabeled sets and cycles convert by diagonalization [4].

- **Unlabeled set.** The conversion is achieved by diagonalization followed by pointing as reflected by the generating function for unlabeled sets:

$$\mathcal{A} = \textsc{Set}(\mathcal{B}) \;\rightarrow\; \Theta\mathcal{A} = \mathcal{A} \cdot \Delta_{\{1\}}\Theta\mathcal{B}$$

The conversion for cardinality restrictions is achieved by recurrence:

$$\mathcal{A} = \textsc{Set}_k(\mathcal{B}) \;\rightarrow$$
$$\Theta\mathcal{A} = \textsc{Set}_{k-1}(\mathcal{B}) \cdot \Delta^{(1)}\Theta\mathcal{B} + \textsc{Set}_{k-2}(\mathcal{B}) \cdot \Delta^{(2)}\Theta\mathcal{B} + \ldots + \textsc{Set}_0(\mathcal{B}) \cdot \Delta^{(k)}\Theta\mathcal{B}$$

where $\textsc{Set}_0(\mathcal{B}) = 1$ and $\textsc{Set}_{\leqslant 0}(\mathcal{B}) = 1$. The base cases for the recurrence are similar to the labeled case: $\textsc{Set}_1(\mathcal{B}) = \mathcal{B}$ and $\textsc{Set}_{\leqslant 1}(\mathcal{B}) = 1 + \mathcal{B}$. Note that for $\textsc{Set}_{\geqslant k}(\mathcal{B})$ we have $\textsc{Set}_{\geqslant 0}(\mathcal{B}) = \textsc{Set}(\mathcal{B})$, which must be expressed in terms of the generalized diagonal:

$$\mathcal{A} = \textsc{Set}_{\geqslant k}(\mathcal{B}) \;\rightarrow$$
$$\Theta\mathcal{A} = \textsc{Set}_{\geqslant k-1}(\mathcal{B}) \cdot \Delta^{(1)}\Theta\mathcal{B} + \textsc{Set}_{\geqslant k-2}(\mathcal{B}) \cdot \Delta^{(2)}\Theta\mathcal{B} + \ldots + \textsc{Set}(\mathcal{B}) \cdot \Delta_{\{j \geq k\}}\Theta\mathcal{B}$$

- **Unlabeled cycle.** Similarly, the conversion is achieved by diagonalization and pointing:

$$\mathcal{A} = \textsc{Cyc}(\mathcal{B}) \;\rightarrow\; \Theta\mathcal{A} = \Delta_{\{\varphi(k)\}}(\Theta\mathcal{B} \cdot \textsc{Seq}(\mathcal{B}))$$

The conversions for cardinality restrictions are similarly derived:

$$\mathcal{A} = \textsc{Cyc}_k(\mathcal{B}) \;\rightarrow\; \Theta\mathcal{A} = \sum_{j \mid k} \Delta_{\{\varphi(j)\delta_{i=j}\}}[\Theta\mathcal{B} \cdot \textsc{Seq}_{k/j-1}(\mathcal{B})]$$
$$\mathcal{A} = \textsc{Cyc}_{\leqslant k}(\mathcal{B}) \;\rightarrow\; \Theta\mathcal{A} = \sum_{j \leq k} \Delta_{\{\varphi(j)\delta_{i=j}\}}[\Theta\mathcal{B} \cdot \textsc{Seq}_{\leqslant \lfloor k/j-1 \rfloor}(\mathcal{B})]$$
$$\mathcal{A} = \textsc{Cyc}_{\geqslant k}(\mathcal{B}) \;\rightarrow\; \Theta\mathcal{A} = \sum_{j < k} \Delta_{\{\varphi(j)\delta_{i=j}\}}[\Theta\mathcal{B} \cdot \textsc{Seq}_{\geqslant \lceil k/j-1 \rceil}(\mathcal{B})] + \Delta_{\{\varphi(j)\delta_{j \geq k}\}}[\Theta\mathcal{B} \cdot \textsc{Seq}(\mathcal{B})]$$

Using the above rules a symbolic equation represented in binary form can be evaluated recursively in linearithmic time. For example, the class of unary-binary trees represented in binary form

```
U = Union(Atom, Union(Product(Atom, U), Product(Product(Atom, U), U)))
```

is first reduced to

```
U = Union(V, W), V = Atom, W = ?
```

`V` is now fully reduced as it evaluates to a terminal rule, while `W` is evaluated recursively:

```
W = Union(X, Y), X = ?, Y = ?
```

`X` and `Y` are evaluated similarly and the binary recursion continues until all rules are fully reduced to Chomsky Normal Form.

Algorithm 1 Convert symbolic equation to standard form

```
1:  procedure CONVERTBASIC
2:      s ← symbolic equation
3:      r ← {}
4:      switch s do
5:          case A = Union(B, C) or Product(B, C)
6:              r ← binary substitution
7:          case A = Sequence(B)
8:              r.add(A = Union(U, V))
9:              r.add(U = Atom(0), V = Product(A, B))
10:         case A = Set(B)
11:             r.add(A = Set(B))
12:             r.add(Theta(A) = Product(A, U), U = Theta(B))
13:         case A = Cycle(B)
14:             r.add(A = Cycle(B))
15:             r.add(Theta(A) = Product(U, V))
16:             r.add(U = Theta(B), V = Sequence(B))
17:     r ← convert sub-rules recursively
18:     return r
```

3.4 Cost Analysis

All of the routines presented above are polynomial in the worst case. Conversion of sequences, sets and cycles requires $O(1)$ substitutions followed by recursive evaluation of the sub-rules. Conversion of cardinality restricted sets (labeled and unlabeled) and cycles (unlabeled) requires $O(k)$ substitutions followed by recursive evaluation of the sub-rules. Conversion of unions and products requires $O(n)$ lookups to avoid redundancies in the standard specification (although this can be reduced to constant time using hash table lookups, we chose to use linear traversals to avoid complicating the code and increasing memory usage), followed by recursive evaluation of the sub-rules. Thus we claim that the overall cost of converting a symbolic equation to standard form is at worst $O(n \log n)$.

4 Generating Coefficients

We derive the counting sequence of a class from its standard specification by tabulating the coefficients for increasing size $k = 0, 1, 2, \ldots$ according to the following protocol [10]:

- $\mathcal{A} = \mathcal{B} + \mathcal{C} \rightarrow a_k = b_k + c_k$
- $\mathcal{A} = \mathcal{B} \times \mathcal{C} \rightarrow a_k = b_{0..k} \times c_{0..k}$ (representing the Cartesian product)
- $\mathcal{A} = \mathcal{Z}^n \rightarrow a_k = \begin{cases} 1 & n = k \\ 0 & n \neq k \end{cases}$

Algorithm 2 Convert cardinality restricted symbolic equation to standard form

```
1: procedure CONVERTRESTRICTED
2:     s ← symbolic equation
3:     r ← {}
4:     switch s do
5:         case A = Sequence(B, card = k)
6:             r.add(exp(B, k)) // binary exponentiation to generate B*B*...*B
7:         case A = Sequence(B, card <= k)
8:             if k > 2 then
9:                 r.add(A = Union(U, V))
10:                r.add(U = Sequence(B, card = k), V = Sequence(B, card <= k-1))
11:            else
12:                r.add(A = Union(U, B), U = Sequence(B, card = k))
13:        case A = Sequence(B, card >= k)
14:            r.add(A = Product(U, V))
15:            r.add(U = Sequence(B, card = k), V = Sequence(B))
16:        case A = Set(B, card = k)
17:            r.add(A = Set(B, card = k))
18:            if k > 2 then
19:                r.add(Theta(A) = Product(U, V))
20:                r.add(U = Theta(B), V = Set(B, card = k-1))
21:            else
22:                r.add(Theta(A) = Product(U, B), U = Theta(B))
23:        case A = Set(B, card <= k)
24:            r.add(A = Set(B, card <= k))
25:            if k > 2 then
26:                r.add(Theta(A) = Product(U, V))
27:                r.add(U = Theta(B), V = Set(B, card = k-1))
28:            else
29:                r.add(Theta(A) = Product(U, V))
30:                r.add(U = Theta(B), V = Atom(0) + B)
31:        case A = Set(B, card >= k)
32:            r.add(A = Set(B, card >= k))
33:            if k > 1 then
34:                r.add(Theta(A) = Product(U, V))
35:                r.add(U = Theta(B), V = Set(B, card = k-1))
36:            else
37:                r.add(Theta(A) = Product(U, V))
38:                r.add(U = Theta(B), V = Set(B))
39:        case A = Cycle(B, card = k)
40:            r.add(A = Cycle(B, card = k))
41:            r.add(Theta(A) = Product(U, V))
42:            r.add(U = Theta(B), V = Sequence(B, card = k-1))
43:        case A = Cycle(B, card <= k)
44:            r.add(A = Cycle(B, card <= k))
45:            r.add(Theta(A) = Product(U, V))
46:            r.add(U = Theta(B), V = Sequence(B, card <= k-1))
47:        case A = Cycle(B, card >= k)
48:            r.add(A = Cycle(B, card >= k))
49:            r.add(Theta(A) = Product(U, V))
50:            r.add(U = Theta(B), V = Sequence(B, card >= k-1))
51:    r ← convert sub-rules recursively
52:    return r
```

Coefficients of theta rules $\Theta\mathcal{R}$ are generated by looking up the corresponding coefficient r of $\mathcal{R}$ and multiplying by k. Additionally, for each coefficient r'_k generated for theta rule $\Theta\mathcal{R}$, the coefficient r'_k/k is tabulated for the corresponding rule $\mathcal{R}$. This is summarized by the following two rules:

- $\mathcal{A} = \Theta\mathcal{B} \rightarrow a_k = k \cdot b_k$
- $\Theta\mathcal{A} = \mathcal{B} \rightarrow a_k = b_k/k$

Recall that delta rules are needed to replace summations of diagonal terms like $A(z^k)$ for unlabeled sets and cycles. The coefficient for the generalized diagonal is entirely computable from pre-tabulated coefficients of $\mathcal{B}$. Diagonal terms of the form $\Delta^{(k)}$ (recall that $\Delta^{(k)}\mathcal{A} = A(z^k)$) are needed to enumerate unlabeled sets with cardinality restrictions and are equivalent to generalized diagonals receiving the Kronecker function as an argument: $\Delta_{\{\delta_{i=k}\}}$. The following rule is used to generate the coefficients:

- $\mathcal{A} = \Delta_{\{u(k)\}}\mathcal{B} \rightarrow a_k = \sum_{i \mid k} u(i) \cdot b_{k/i}$

Recall from Section 2 that the exponential generating function for a size n labeled class is normalized by $n!$. The correct counting sequence for the labeled class $a_n = n![z^n]A(z)$ is thus obtained by multiplying the tabulated coefficients by $k!$ for each value of k in the table. The tabulated coefficients for unlabeled binary trees (**Figure 6**) yield the counting sequence $0, 1, 1, 2, 5, 14, \ldots$ The corresponding counting sequence for labeled binary trees would be $0, 1, 2, 12, 120, 1680, \ldots$

k	u_k	v_k	b_k
0	0	0	0
1	1	0	1
2	0	1	1
3	0	2	2
4	0	5	5
5	0	14	14

Figure 6. Table of coefficients for binary trees.

All non-terminal rules in the standard specification depend on one or more sub-rules. For example, $\mathcal{A} = \mathcal{B} + \mathcal{C}$ depends on $\mathcal{B}$ and $\mathcal{C}$. Thus a coefficient a_k cannot be generated until b_k and c_k have already been generated. There are several ways to resolve this. The first is to sort the standard specification so that the coefficients are generated in the proper order. For basic objects it suffices to sort the rules by decreasing order of *valuation* (the concept of valuation is discussed in the next section); however, we found that for more complex objects involving sets, cycles, cardinality restrictions and/or unlabeled grammars the sorting rules are less straightforward, resulting in a significantly more complicated implementation. Furthermore,

the cost to sort the rules would be a major performance factor for large symbolic specifications. The alternative method is to generate coefficients recursively, so a call to generate a_k would first recursively execute calls to generate b_k and c_k:

$$\texttt{generate}(\mathcal{A}, k) = \texttt{generate}(\mathcal{B}, k) + \texttt{generate}(\mathcal{C}, k)$$

where $\texttt{generate}(\mathcal{R}, k)$ returns r_k if it exists and recursively executes on the sub-rules of $\mathcal{R}$ otherwise. This method does not require the standard specification to be ordered in any specific way. There is a possibility of infinite recursion if, for example, $\mathcal{A}$ depends on $\mathcal{B}$ and $\mathcal{B}$ (or one of $\mathcal{B}$'s children) depends on $\mathcal{A}$. This is preventable by passing a collection of values down the recursion tree where each rule is the parent of its sub-rules. Each parent visited by a recursive call first adds itself to the collection and then passes the collection down to its children. Thus we can determine if a rule $\mathcal{R}$ occurs in the subtree rooted by $\mathcal{R}$ for the same value of k and prevent recursion on $\mathcal{R}$.

4.1 Valuations

All nontrivial standard specifications contain cyclic dependencies. Consider the standard specification for binary trees, where $\mathcal{V}$ depends on $\mathcal{B}$ but $\mathcal{B}$ also depends on $\mathcal{V}$. We resolve these dependencies by observing that many rules have coefficients of zero for objects of some size k or less. The *valuation* of a rule is thus defined to be the value of k below which the coefficient of the rule is zero:

$$val(\mathcal{R}) = v \rightarrow r_{k<v} = 0$$

The valuation is used as a base case to avoid cyclic dependencies and can be computed by a simple iterative routine [10] (**Algorithm 4**). Each iteration updates the valuations for all of the standard form rules until a stable configuration is reached, i.e. none of the valuations change between two consecutive iterations.

- $\mathcal{A} = \mathcal{B} + \mathcal{C} \rightarrow val_j(\mathcal{A}) = \texttt{min}(val_{j-1}(\mathcal{B}), val_{j-1}(\mathcal{C}))$
- $\mathcal{A} = \mathcal{B} \cdot \mathcal{C} \rightarrow val_j(\mathcal{A}) = val_{j-1}(\mathcal{B}) + val_{j-1}(\mathcal{C})$

The valuation of an atom is equal to its size and does not change between iterations. We determined through correspondence with Zimmermann that a theta rule $\Theta\mathcal{R}$ receives the same valuation as the corresponding rule $\mathcal{R}$. Similarly, a delta rule $\Delta\mathcal{R}$ receives the same valuation as the corresponding rule $\mathcal{R}$. This gives us the following additional rules:

- $\mathcal{A} = \mathcal{Z} \rightarrow val_j(\mathcal{A}) = val_{j-1}(\mathcal{A})$
- $\mathcal{A} = \Theta\mathcal{B} \rightarrow val(\mathcal{A}) = val(\mathcal{B})$
- $\mathcal{A} = \Delta_{\{u(k)\}}\mathcal{B} \rightarrow val(\mathcal{A}) = val(\mathcal{B})$

We found through implementation and testing of the routine that Chomsky Normal Form is not sufficient to completely eliminate base case issues. Consider the standard form conversion for labeled sets:

$$\mathcal{A} = \textsc{Set}(\mathcal{B}) \rightarrow \Theta\mathcal{A} = \mathcal{A} \cdot \Theta\mathcal{B}$$

We cannot compute the valuation of this rule using the method defined above due to its self-recursive nature. Because $val(\mathcal{A}) = val(\Theta\mathcal{A})$, the valuation would be $val_j(\mathcal{A}) = val_{j-1}(\mathcal{A}) + val_{j-1}(\mathcal{B})$, which is unsolvable. Instead we observe that every set $\mathcal{A} = \textsc{Set}(\mathcal{B})$ contains the single element ε for size $k = 0$, regardless of $\mathcal{B}$. Therefore the valuation of an unrestricted set is always 0 and there is a single set of size 0, i.e. $a_0 = 1$. Similarly, we observe that every cycle $\mathcal{A} = \textsc{Cyc}(\mathcal{B})$ contains no elements for size $k = 0$ (a cycle of size 0 does not exist) and a single $\mathcal{B}$ for size $k = 1$. Therefore the valuation of an unrestricted cycle is always 1. This gives us the following rules:

- $\mathcal{A} = \textsc{Set}(\mathcal{B}) \rightarrow val(\mathcal{A}) = 0$
- $\mathcal{A} = \textsc{Cyc}(\mathcal{B}) \rightarrow val(\mathcal{A}) = 1$

It is essential that these set and cycle rules be included in the standard specification so that the valuations can be computed. The above valuation rules apply to sets and cycles with $\leq$ cardinality restrictions as well because these objects include $k = 0$ and $k = 1$. For sets and cycles with $=$ and $\geq$ restrictions, however, there is a higher minimum cardinality and this is reflected by the valuations:

- $\mathcal{A} = \textsc{Set}_{\leqslant k}(\mathcal{B}) \rightarrow val(\mathcal{A}) = 0$
- $\mathcal{A} = \textsc{Set}_{k}(\mathcal{B}) \rightarrow val(\mathcal{A}) = k \cdot val(\mathcal{B})$
- $\mathcal{A} = \textsc{Set}_{\geqslant k}(\mathcal{B}) \rightarrow val(\mathcal{A}) = k \cdot val(\mathcal{B})$
- $\mathcal{A} = \textsc{Cyc}_{\leqslant k}(\mathcal{B}) \rightarrow val(\mathcal{A}) = 1$
- $\mathcal{A} = \textsc{Cyc}_{k}(\mathcal{B}) \rightarrow val(\mathcal{A}) = k \cdot val(\mathcal{B})$
- $\mathcal{A} = \textsc{Cyc}_{\geqslant k}(\mathcal{B}) \rightarrow val(\mathcal{A}) = k \cdot val(\mathcal{B})$

The valuation rules described in this section apply to both labeled and unlabeled objects. Implemented as an iterative routine, they provide all of the necessary base cases for recursive coefficient generation. For example, given the standard specification for rooted unlabeled trees:

$$\mathcal{G} = \mathcal{A} \cdot \mathcal{B},\ \mathcal{A} = \mathcal{Z},\ \mathcal{B} = \textsc{Set}(\mathcal{G}),\ \Theta\mathcal{B} = \mathcal{B} \cdot \mathcal{C},\ \mathcal{C} = \Delta_{\{1\}}\mathcal{D},\ \mathcal{D} = \Theta\mathcal{G}$$

the following valuations are computed:

$$val(\mathcal{G}) = 1,\ val(\mathcal{A}) = 1,\ val(\mathcal{B}) = 0,\ val(\mathcal{C}) = 1,\ val(\mathcal{D}) = 1$$

The same iterative routine for determining rule valuations can be used to check the validity of a user inputted symbolic equation, as an equation with unresolvable cyclic dependencies will not terminate with a set of rule valuations. This allows us to catch and handle user errors gracefully.

4.2 Implementation

The purpose of **Algorithm 3** is to generate coefficients from a standard specification. For increasing values of size $k = 0, 1, 2, 3, \ldots$ the coefficients are evaluated and tabulated in a table t. The table t can be implemented as a hash table of hash tables for fast lookups and modifications.

As discussed above,the process of generating coefficients for certain rules is dependant on the coefficients of their sub-rules. Therefore it is important to evaluate the coefficients of the sub-rules first. Rather than attempt to sort the rules in the correct order, we generate the coefficients recursively. To avoid infinite recursion resulting from cyclic dependencies, **Algorithm 4** computes the valuation of each rule as defined by Zimmermann and stores all of the valuations in a hash table. The table of valuations can then be passed into **Algorithm 3** and used to handle the initial values of coefficients.

4.3 Cost Analysis

All of the routines presented in this section cost polynomial in the worst case. **Algorithm 3** costs $O(1)$ for atom, $O(1)$ for union and $O(1)$ for theta rules. The cost to evaluate delta and Cartesian product rules is $O(k)$. Clearly the overall cost to tabulate n rules is at must $O(nk)$.

Algorithm 4 continues to iterate until none of the valuations change from iteration j to iteration $j+1$. Each iteration involves at most $O(n)$ additions or products or $O(n)$ `min` calculations each costing $O(1)$. It can be observed that for rule $\mathcal{R}$ and iteration j of the routine, $val_{j-1}(\mathcal{R}) \leq val_j(\mathcal{R})$ and since the minimum valuation any rule can take is zero, the routine necessarily converges if given a valid input [10].

```
> sym = Sequence('F', Union(Atom(1), Atom(2)))
F = Sequence(Z^1 + Z^2)
> rules = ConvertToStandardForm(sym)
A = D + E
C = Z^0
B = F * A
E = Z^2
D = Z^1
F = C + B
> val = ComputeRuleValuations(rules)
[('A', 1), ('C', 0), ('B', 1), ('E', 2), ('D', 1), ('F', 0)]
> EnumerateFromStandardForm(rules, 10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Figure 7. Enumerating the Fibonacci sequence: $\mathcal{F} = \textsc{Seq}\left(\mathcal{Z}^1 + \mathcal{Z}^2\right)$.

Algorithm 3 Generate coefficients

```
1: procedure GENERATE(k)
2:     rules ← standard specification
3:     t ← table of coefficients
4:     for r in rules do
5:         t[k][r] ← EVALUATE(k, r)
6:     return t
7:
8: procedure EVALUATE(k, r)
9:     t ← table of coefficients
10:    v ← table of valuations
11:    if r in t then
12:        return t[k][r]
13:    if k < v[r] then
14:        return 0
15:    switch r do
16:        case A = Union(B, C)
17:            return EVALUATE(k, B) + EVALUATE(k, C)
18:        case A = Product(B, C)
19:            ℓ, i ← 0
20:            while i ≤ k do
21:                ℓ ← ℓ + EVALUATE(i, B) * EVALUATE(k − i, C)
22:                i ← i + 1
23:            return ℓ
24:        case A = Atom(N)
25:            if N = k then
26:                return 1
27:            else
28:                return 0
29:        case A = Theta(B)
30:            return k * EVALUATE(k, B)
31:        case A = Delta(f, B)
32:            ℓ ← 0
33:            i ← 1
34:            while i ≤ k do
35:                if k % i = 0 then
36:                    ℓ ← ℓ + f(i) * EVALUATE(k/i, B)
37:                i ← i + 1
38:            return ℓ
39:    return t
```

Algorithm 4 Compute rule valuations

```
1: procedure INITIALIZERULEVALUATIONS
2:     rules ← standard specification
3:     v ← {}
4:     for r in rules do
5:         switch r do
6:             case A = Atom(N)
7:                 v[A] ← N
8:             case A = Set(B) or Set(B, card <= k)
9:                 v[A] ← 0
10:            case A = Cycle(B) or Cycle(B, card <= k)
11:                v[A] ← 1
12:            case default
13:                v[A] ← ∞
14:    return v
15:
16: procedure COMPUTERULEVALUATIONS
17:    rules ← standard specification
18:    v ← table of valuations
19:    done ← True
20:    for r in rules do
21:        prev ← v[A]
22:        switch r do
23:            case A = Union(B, C)
24:                v[A] ← min(v[B], v[C])
25:            case A = Product(B, C)
26:                v[A] ← v[B] + v[C]
27:            case A = Set(B, card = k) or Set(B, card >= k)
28:                v[A] ← k * v[B]
29:            case A = Cycle(B, card = k) or Cycle(B, card >= k)
30:                v[A] ← k * v[B]
31:            case A = Theta(B)
32:                v[A] ← v[B]
33:            case A = Delta(B)
34:                v[A] ← v[B]
35:            case default
36:                continue
37:        if v[A] ≠ prev then
38:            done ← False
39:    if done then
40:        return v
41:    else
42:        return COMPUTERULEVALUATIONS
```

```
> sym = Cycle('T0', KSequence(Atom(1), ">=", 1))
T0 = Cycle(Sequence(Z^1, card >= 1))
> rules = ConvertToStandardForm(sym, labeled=False)
T8 = T4 + T9      T2 = T4 + T3
T9 = T8 * T7      T3 = T2 * T1
T6 = T2 * T5      Theta(T0) = Delta(T6)
T7 = Z^1          T1 = T7 * T8
T4 = Z^0          T0 = Cyc(T1)
T5 = Theta(T1)
> val = ComputeRuleValuations(rules)
{'T8': 0, 'T9': 1, 'T6': 1, 'T7': 1, 'T4': 0, 'T5': 1,
 'T2': 0, 'T3': 1, 'T0': 1, 'T1': 1}
> EnumerateFromStandardForm(rules, 10)
[0, 1, 2, 3, 5, 7, 13, 19, 35, 59, 107]
```

Figure 8. Enumerating unlabeled necklaces: $\mathcal{T}_0 = \textsc{Cyc}\left(\textsc{Seq}_{\geqslant 1}(\mathcal{Z})\right)$.

5 Conclusion

To date, there exists no open source, user-friendly tool for combinatorial enumeration. We have developed a fully functional Python library (**Figure 7, 8**) based on the methods discussed in this paper. We have also produced the first paper which fully documents all of the algorithms underlying this method of combinatorial enumeration through a combination of definitions [3] [6], algorithms [10] [5] and our own observations drawn from building and testing the prototype.

The Python library has been extensively tested on both labeled and unlabeled grammars composed of union, product, sequence, set and cycle constructions with and without cardinality restrictions. The Online Encyclopedia of Integer Sequences [9] and Maple's `combstruct` library were used as references for testing. Immediate next steps will be to extend support to the more mathematically complex undirected sequence and undirected cycle constructions.

5.1 Future Work

The end goal of this project is to have a fast, extendable and fully accessible code base. To that end, in addition to extending the code to support additional constructions, our next steps will include redeveloping the code in Go for better performance and deployability.

All of the protocols implemented in this work operate in polynomial time. Before we conclude it must be noted that a logarithmic algorithm for enumeration was introduced in 2012 [8]. Why are we implementing a polytime algorithm when a faster algorithm has been developed? First, the algorithm is highly complex and the framework of formal algebra required would prevent this library from being standalone, which was one of the original goals of the project. Second, we determined that polytime performance did not significantly affect usage in our test cases, although we accept that some applications of combinatorial enumeration

can involve very large inputs. Given these considerations and the general lack of clarity in the literature of this subject area, we decided to prioritize developing a simple, understandable and easily extendable algorithm as an accessible tool for researchers over faster performance. Therefore we leave the implementation of the logarithmic algorithm to future work.

6 Acknowledgements

We thank Dr. Paul Zimmermann for assistance via email.

References

[1] Maryam Bahrani and Jérémie Lumbroso. Enumerations, forbidden subgraph characterizations, and the split-decomposition. *arXiv*, 2016.

[2] Alexandre Casamayou, Nathann Cohen, et al. Calcul Mathématique avec Sage. 2013.

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

[4] Philippe Flajolet, Paul Zimmermann, and Bernard van Cutsem. A calculus of random generation: unlabelled structures. 1993.

[5] Philippe Flajolet, Paul Zimmermann, and Bernard van Cutsem. A Calculus for the Random Generation of Labelled Combinatorial Structures. *Theoretical Computer Science*, 132(1-2):1–35, 1994.

[6] Jérémie Lumbroso and Basile Morcrette. A Gentle Introduction to Analytic Combinatorics. *Proceedings of the Nablus2014 CIMPA Summer School*, 2014.

[7] Alasdair McAndrew. Mupad. *Linux J.*, 1999(63es), July 1999.

[8] Carine Pivoteau, Bruno Salvy, and Michele Soria. Algorithms for Combinatorial Systems: Well-Founded Systems and Newton Iterations. *arXiv*, 2012.

[9] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. 1964.

[10] Paul Zimmermann. *Séries génératrices et analyse automatique d'algorithmes*. PhD thesis, École Polytechnique, 1991.

[11] Paul Zimmermann. Gaïa: A Package for the Random Generation of Combinatorial Structures. *MapleTech*, 1(1):38–46, 1994.

This paper represents my own work in accordance with University regulations.