Efficient iteration in admissible combinatorial classes

Conrado Martínez∗, Xavier Molinero

Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya,
E-08034 Barcelona, Spain

Abstract

The exhaustive generation of combinatorial objects has a vast range of practical applications and is a common theme in the combinatorial research field. But most previous works in this area concentrate in the efficient generation of particular families of combinatorial objects. The novel approach of the work presented here is to provide efficient generic algorithms, where the input is not just the size n of the objects to be generated but a finite specification of the combinatorial class whose objects we want to list. Since the algorithms are generic, they do not exploit any particular feature of the class to be generated; nevertheless, they work in constant amortized time per generated object, that is, they generate all N objects of a given size in Θ(N) time. These algorithms are useful for both rapid prototyping and for inclusion into general purposes libraries because of their flexibility, with only a relatively modest penalty on efficiency. Furthermore, the framework presented in this paper nicely combines with the framework developed by Flajolet et al. for the enumeration and random generation of combinatorial objects, and with the framework developed by the authors for the unranking of combinatorial objects.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Generation of combinatorial objects; Iteration in combinatorial classes; Unranking; Generation of combinatorial objects in lexicographic order; Constant amortized time

∗ This research was supported by the Future and Emergent Technologies programme of the EU under contract IST-1999-14186 (ALCOM-FT) and the Spanish Min. of Science and Technology project TIC2002-00190 (AEDIPI).

∗ Corresponding author. Tel.: +34 934137849; fax: +34 934137833.
E-mail addresses: conrado@lsi.upc.edu (C. Martínez), molinero@lsi.upc.edu (X. Molinero).

0304-3975/S - see front matter © 2005 Elsevier B.V. All rights reserved.
1. Introduction

The exhaustive generation of all objects of a given size in a combinatorial class has been the subject of intense research for already four decades (see for instance [10,16,17,22]). Many efforts are still devoted in the quest for efficient algorithms for combinatorial classes for which no such algorithms are known, as well as to the improvement of existing algorithms. Among the numerous researchers that have been attracted by this important topic was Alberto del Lungo [2,3,11], to whom we dedicate this paper.

Efficient algorithms exist for the generation of permutations, trees, strings, cycles of \( k \) beads, partitions, subsets of \( k \) elements out of \( n \), etc. (see, for instance, [10,16,18]). Although it is not always possible to attain it, the ultimate goal in efficiency is to generate all \( N \) objects of size \( n \) of a given class in time proportional to \( N \); in other words, to achieve constant amortized time (CAT) per generated object. Sometimes it is desired to generate the objects in a particular order, for instance, permutations in lexicographic order or bitstrings in Gray order, but in many other situations we can choose whatever generation order yields the best performance. But in any case the vast majority of results in this area concern particular combinatorial classes or more or less restricted families of them.

In this paper we follow a rather different approach, since our goal is to develop a generic algorithm to solve the exhaustive generation problem when the class of the objects to be generated is a parameter itself. Other generic approaches, such as the ECO method (see for instance [1–3,11]) have been proposed, but they differ substantially with the one presented here, each approach imposing different requirements on the classes that can be generated, the way to express them, etc. In particular, our algorithms need to be able to count how many objects there are in a given combinatorial class, hence the combinatorial classes we can deal with are those built using the so-called admissible operators. This approach was fully exploited by Flajolet et al. [5], with their celebrated application to the random generation of combinatorial objects. Later, the authors of this paper showed that this approach can also be successfully used for the unranking of combinatorial objects, that is, given a specification for a combinatorial class, a size \( n \) and a rank \( i \), generate the \( i \)th object of size \( n \) in the given class [13].

One of the main contributions of this paper is to show the viability of this generic approach to cope with the exhaustive generation problem, providing an elegant, effective and efficient solution. Examples of combinatorial classes which can be exhaustively generated using our approach include permutations, functional graphs, integer partitions, integer compositions, surjections, regular languages, simply generated families of trees (including binary trees, Cayley trees and plane general trees, among others), Dyck paths, etc. For instance, to list all partitions of the number \( n \) we could supply our algorithm with the specification

\[
P = \text{MSet}(Z \times \text{Seq}(Z))
\]

and ask it to exhaustively generate all objects of size \( n \) in the class \( P \). The specification above simply says that an integer partition is a multiset—repetitions are thus allowed—of integers; each integer is represented by an atom \((Z)\) followed by a sequence of atoms.
(\text{Seq}(\mathbb{Z})), in other words, it is represented by a sequence of at least one atom. The length or size of such a sequence coincides with the represented integer, and the size of the partition is hence the sum of the sizes of its components. Another example is the generation of all words of some given size in a regular language, for instance, the language $L = \{w \in (a + b)^* \mid w \text{ does not contain two consecutive } a's\}$. We can express this language using the specification

$$L = M + M \times a,$$

$$M = \varepsilon + M \times b + M \times a \times b$$

and then supply this specification to our algorithm and request the complete list of objects in $L$ of size $n$. In the equations above, $\varepsilon$ represents the class that only contains an object of size 0, $+$ denotes disjoint unions, $\times$ is the usual Cartesian product, and $a$ and $b$ denote atomic classes each containing a single object of size 1. Any regular language can be unambiguously be described in a similar way as this toy example, and hence exhaustively generated using our algorithms. It is also possible to exhaustively generate all words of a given size in a context-free language, provided that we find a suitable unambiguous grammar for the language.

Since the algorithms described in this work are generic, they do not take advantage of particular features of the combinatorial class whose objects we want to generate. However, the algorithms achieve constant amortized time per generated object, and thus they are competitive when compared with algorithms specifically designed for the corresponding class. The fact that our algorithms exhibit CAT performance is one of the main results in this paper.

Most likely, the price of being general is that the involved constant factors are a bit larger than in their ad-hoc counterparts, but the flexibility of our generic algorithms makes them attractive for rapid prototyping and for their inclusion in general combinatorial libraries, like the combinstruct package for MAPLE [4,23], or the MuPAD-combinat package [8].

The interest that such generic algorithms can have is well exemplified by the following situation: the combinstruct package provides efficient routines for counting and random generation given a finite specification of a combinatorial class; it also provides a routine allstructs which produces a list of all combinatorial objects of a given size. But, except for a few particular cases where allstructs calls a specialized efficient algorithm, the task is performed by repeatedly drawing objects at random until all them have been generated. Of course, this is useless if the objects have to be processed in some specific order or different executions have to generate the objects in the same order. Furthermore, it is usually not as efficient as using an algorithm specifically designed for exhaustive generation, like the ones in this paper.

This paper is organized as follows. In Section 2, we introduce the basic concepts and notation needed in the following sections. Sections 3 and 4 present the algorithms for the generation of labelled and unlabelled classes, respectively. In Section 5 we analyze the performance of our algorithms. Preliminary descriptions of the results in this paper have appeared in [12,14].
2. Admissible classes, orders and iterators

2.1. Admissible classes

A key aspect in the design of our generic iteration algorithms is the ability to determine how many elements of a given size there are in the given combinatorial class. Given a combinatorial class \( \mathcal{A} \), we denote by \( \mathcal{A}_n = \{ x \in \mathcal{A} \mid |x| = n \} \) the subset of objects in \( \mathcal{A} \) of size \( n \). We use the corresponding lowercase letters to denote the cardinality of such subset. Thus \( a_n = \# \mathcal{A}_n \) is the number of objects in \( \mathcal{A}_n \).

Since the ability to count the number of objects in \( \mathcal{A}_n \) is crucial in our algorithms, we will only consider \textit{admissible} combinatorial classes. Admissible combinatorial classes include the \( e \)-class (which only contains an element of size 0, denoted \( e \)), atomic classes (those that only contain an element of size 1) and those inductively built using \textit{admissible operators} on admissible classes (see for instance [6,20,21]).

\textbf{Definition 1.} Let \( \Psi \) be an operator on combinatorial classes \( \mathcal{A}_1, \ldots, \mathcal{A}_k \). The operator \( \Psi \) is \textit{admissible} if and only if there exists an operator \( \Phi \) over the generating functions \( A_1(z), \ldots, A_k(z) \) corresponding to the classes \( \mathcal{A}_1, \ldots, \mathcal{A}_k \) such that the generating function \( C(z) \) of the class \( \mathcal{C} = \Psi(\mathcal{A}_1, \ldots, \mathcal{A}_k) \) satisfies

\[ C(z) = \Phi(A_1(z), \ldots, A_k(z)), \]

where the generating function \( C(z) \) of any unlabelled class \( \mathcal{C} \) is

\[ C(z) = \sum_{n \geq 0} c_n z^n = \sum_{\gamma \in \mathcal{C}} z^{\|\gamma\|}, \]

whereas the generating function for a labelled class \( \mathcal{C} \) is

\[ C(z) = \sum_{n \geq 0} \frac{c_n}{n!} z^n = \sum_{\gamma \in \mathcal{C}} \frac{z^{\|\gamma\|}}{n!}. \]

Examples of admissible operators include disjoint union, Cartesian products, sequences, sets, etc. However, other common combinatorial constructs, like difference and intersections, are not admissible.

We will concentrate in a fixed collection of admissible operators. For unlabelled combinatorial classes, where the atoms that compose the objects in the class are not distinguishable,\(^1\) we will consider classes built using disjoint unions (‘+’), Cartesian products (‘\( \times \)’), sequences of objects (‘Seq’), cycles of objects (‘Cycle’), powersets of objects (‘PSet’) and multisets of objects (‘MSet’). For labelled combinatorial classes, where the atoms of each object of size \( n \) bear distinct labels in \([1..n]\), we consider disjoint unions (‘+’), labelled or partitional products (‘\( \star \)’), sequences of objects (‘Seq’), cycles of objects (‘Cycle’) and sets of objects (‘Set’). Since each atom carries a distinct label the notion of a labelled multiset makes no sense.

\(^1\) We can have an unlabelled object like the multiset \( \{ a, a, a, b, c \} \) with 5 atoms, where there are 3 “distinct” atoms or symbols (\( a, b, c \)), but we cannot distinguish between the three \( a \)’s, for instance.
Other admissible operators, like substitutions or sequences, cycles, sets, powersets and multisets with restricted number of components, can also be easily accommodated in our framework. We have actually designed, analyzed and implemented algorithms to cope with classes built using these other operators, but we will not explain them in detail.

Even though we are considering a relatively small number of combinatorial operators, a large number of important combinatorial classes can be expressed in terms of these operators, including permutations, Cayley trees, all sorts of simply generated families of trees, surjections, functional graphs, partitions, etc. Let us formalize now the notion of an admissible combinatorial specification.

**Definition 2.** An admissible combinatorial specification is a finite set of equations of the form

\[
A_1 = \Psi_1(A_{1,1}, A_{1,2}, \ldots, A_{1,n_1}), \\
A_2 = \Psi_2(A_{2,1}, A_{2,2}, \ldots, A_{2,n_2}), \\
\ldots \\
A_k = \Psi_k(A_{k,1}, A_{k,2}, \ldots, A_{k,n_k}),
\]

where each \(A_{i,j}\) is either \(\varepsilon\), an atom (generically denoted \(Z\)), or the left-hand side \(A_\ell\) of one of the equations, for some \(\ell, 1 \leq \ell \leq k\). Furthermore, the symbols \(A_\ell\) in the left-hand side of the equations are all distinct, and all the operators \(\Psi_\ell\) belong to the set \(\{+, \times, \text{Seq}, \text{Cycle}, \text{PSet}, \text{MSet}\}\), in the case of unlabelled specifications, or to the set \(\{+, \ast, \text{Seq}, \text{Cycle}, \text{Set}\}\), in the case of labelled specifications.

The restriction of cardinalities (number of components) of the labelled operators \(\text{Seq, Cycle, Set}\) and unlabelled operators \(\text{Seq, Cycle, PSet, MSet}\) is also admissible, i.e., operators of the form \(X(\cdot, \text{card} = k)\), \(X(\cdot, \text{card} < k)\) and \(X(\cdot, \text{card} > k)\) are admissible for any constant \(k\).

So a combinatorial class \(\mathcal{A}\) is admissible if it is an \(\varepsilon\)-class, an atomic class or there exists an admissible combinatorial specification such that \(\mathcal{A}\) is the left-hand side of one of the equations.

For instance, the class \(\mathcal{B}\) of (unlabelled) binary trees is admissible since it can be specified by

\[
\mathcal{B} = Z + \mathcal{P}, \\
\mathcal{P} = \mathcal{B} \times \mathcal{B},
\]

which is customarily written \(\mathcal{B} = Z + \mathcal{B} \times \mathcal{B}\). As another example, a functional graph is a set of cycles of Cayley trees, so one possible specification for the class \(\mathcal{F}\) could be

\[
\mathcal{F} = \text{Set}(\mathcal{C}), \\
\mathcal{C} = \text{Cycle}(\mathcal{T}), \\
\mathcal{T} = Z \ast \mathcal{S}, \\
\mathcal{S} = \text{Set}(\mathcal{T}),
\]

or more briefly, \(\mathcal{F} = \text{Set}(\text{Cycle}(\mathcal{T}))\), \(\mathcal{T} = Z \ast \text{Set}(\mathcal{T})\). Fig. 3 in page 10 gives an example of a functional graph of size 9. Other examples of admissible combinatorial classes and their specifications are given in Table 1.
work for any admissible class \( \mathcal{A} \) and a size \( n \), we can efficiently compute \( a_n \) [4] by means of a routine \texttt{count} which receives a finite specification of the class \( \mathcal{A} \) and the size \( n \).

### 2.2. Ordering the objects

Another key aspect in the design and analysis of the algorithms to generate all the objects in \( \mathcal{A}_n \) is the choice of the order that we shall use to generate the objects.

We want our algorithms to be deterministic so that they always produce the same list of objects given an admissible specification for \( \mathcal{A} \) and a size \( n \). Since our algorithms should work for any admissible class \( \mathcal{A} \), what we need now is to provide an inductive definition of the order \( \prec_{\mathcal{A}_n} \) among the objects of \( \mathcal{A}_n \), in terms of the orders of the classes and operators used to build \( \mathcal{A}_n \). For the rest of the paper, the rank of an object \( z \) in \( \mathcal{A}_n \) is the number of objects that precede \( z \) in \( \mathcal{A}_n \), according to \( \prec_{\mathcal{A}_n} \). Thus the first object has rank 0 and the last one has rank \( a_n - 1 \).

The order for the \( \kappa \)-class and atomic classes is trivial as they only contain one object. For disjoint unions (either labelled or unlabelled), if \( \mathcal{C} = \mathcal{A} + \mathcal{B} \) then we say that, for any \( n \), the objects in \( \mathcal{A}_n \) come first, then the objects in \( \mathcal{B}_n \). More formally, if \( \gamma \) and \( \gamma' \) are two objects in \( \mathcal{C}_n \) then \( \gamma \prec_{\mathcal{C}_n} \gamma' \) if and only if \( \gamma \in \mathcal{A}_n \) and \( \gamma' \in \mathcal{B}_n \), or both belong to \( \mathcal{A}_n \) and \( \gamma \prec_{\mathcal{A}_n} \gamma' \), or both belong to \( \mathcal{B}_n \) and \( \gamma \prec_{\mathcal{B}_n} \gamma' \). It is important to point out here that even though \( \mathcal{A} + \mathcal{B} \equiv \mathcal{B} + \mathcal{A} \) the order induced by these two isomorphic specifications is not the same. In general, the order in a class \( \mathcal{C} \) is not dictated by the combinatorial structure of \( \mathcal{C} \) itself, but by the specification that we have used—and there will be many different but equivalent ways to specify a given class.

Things get more interesting when we consider products. In the case of unlabelled products \( \mathcal{C} = \mathcal{A} \times \mathcal{B} \), if we have two objects \( \gamma = (\alpha, \beta) \) and \( \gamma' = (\alpha', \beta') \) such that \( |\alpha| = |\alpha'| = j \) and \( |\beta| = |\beta'| = n - j \) then it is natural to use the lexicographic criterion to order them: \( \gamma \prec_{\mathcal{C}_n} \gamma' \) if \( \alpha \prec_{\mathcal{A}_j} \alpha' \), or if \( \alpha = \alpha' \) and \( \beta \prec_{\mathcal{B}_{n-j}} \beta' \). But we need also to define the order when \( |\alpha| \neq |\alpha'| \). The most obvious solution, the \textit{lexicographic order}, is induced by the specification

\[
\mathcal{C}_n = \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \mathcal{A}_2 \times \mathcal{B}_{n-2} + \cdots + \mathcal{A}_n \times \mathcal{B}_0,
\]

although other “exotic” choices are possible, like the \textit{boustrophedonic order} [5,13] induced by the specification

\[
\mathcal{C}_n = \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_n \times \mathcal{B}_0 + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \mathcal{A}_{n-1} \times \mathcal{B}_1 + \cdots.
\]
For the generation problem, however, the boustrophedonic order yields no performance
improvement if compared to the lexicographic order; hence, we shall only consider the
lexicographic order, formally defined as follows: \( \gamma = (\alpha, \beta) \prec_{C_n} \gamma' = (\alpha', \beta') \) if and only if
\(|x| < |x'|\), or \(|x| = |x'| = j\) and \(\alpha \prec_{A_j} \alpha'\), or \(\alpha = \alpha'\) and \(\beta \prec_{B_{n-j}} \beta'\).

When labelled products are considered we need to take also into account the labels of
the atoms. Recall that given two labelled objects \(\alpha\) and \(\beta\) of sizes \(j\) and \(n - j\), respectively,
their labelled product is a set of \(\binom{n}{j}\) labelled objects of size \(n\) which result from the \(\binom{n}{j}\)
consistent relabellings of the pair \((\alpha, \beta)\) so that each atom of the pair has a distinct label in
the range \([1..n]\) while respecting the order induced by the original labels of \(\alpha\) and \(\beta\). For
example, if \(\alpha = 132\) and \(\beta = 21\) (these two objects belong to the labelled class \(\text{Seq}(Z)\),
i.e., permutations) then\(^2\)

\[ \alpha \star \beta = \{13254, 14253, 14352, \ldots, 35421\}. \]

Therefore we can write

\[ A \star B = \bigcup_{x \in A, \beta \in B} \alpha \star \beta. \]

Given a partition \(\rho\) of \([1..n]\) into a \(j\)-subset \(\{\ell_1, \ldots, \ell_j\}\) and a \((n - j)\)-subset \(\{\ell'_{n-j}, \ldots, \ell'_1\}\),
we denote \((\alpha, \beta, \rho)\) the labelled object of size \(n\) which results when we relabel the atoms of
the pair \((\alpha, \beta)\) according to \(\rho\). For example, if \(\alpha = 21\), \(\beta = 213\) and \(\rho = \{1, 3\}, \{2, 4, 5\}\)
then \((\alpha, \beta, \rho) = 31425\). We say that \(\rho\) is a \((n, j)\)-partition. Let \(S_{n,j}\) denote the set of all
possible \((n, j)\)-partitions and assume that it is equipped with some well-defined order \(\prec_{S_{n,j}}\).

Two different orders for the objects in \(A_j \star B_{n-j}\) arise then in a natural way stemming from the
following specifications:

\[ A_j \star B_{n-j} = \bigcup_{\rho \in S_{n,j}} \bigcup_{\alpha \in A_j} \bigcup_{\beta \in B_{n-j}} (\alpha, \beta, \rho) \tag{1} \]

and

\[ A_j \star B_{n-j} = \bigcup_{\rho \in S_{n,j}} \bigcup_{\alpha \in A_j} \bigcup_{\beta \in B_{n-j}} (\alpha, \beta, \rho). \tag{2} \]

If we use the specification (1) for \(C = A \star B\), we have that for two objects \(\gamma\) and \(\gamma'\)
of size \(n\), \(\gamma = (\alpha, \beta, \rho) \prec \gamma' = (\alpha', \beta', \rho')\) if \(|x| < |x'|\), or \(|x| = |x'| = j\) and \(\alpha \prec_{A_j} \alpha'\),
or \(\rho = \rho'\) and \(\alpha \prec_{A_j} \alpha'\), or \(\rho = \rho'\) and \(\alpha = \alpha'\) and \(\beta \prec_{B_{n-j}} \beta'\). On the other hand, if
we use the specification (2) we have that \(\gamma = (\alpha, \beta, \rho) \prec_{C_n} \gamma' = (\alpha', \beta', \rho')\) if \(|x| < |x'|\),
or \(|x| = |x'| = j\) and \(\alpha \prec_{A_j} \alpha'\), or \(\alpha = \alpha'\) and \(\beta \prec_{B_{n-j}} \beta'\), or \(\alpha = \alpha'\) and \(\beta = \beta'\)
and \(\rho \prec_{S_{n,j}} \rho'\). We will use the name structure-first (or SF for short) for the first order, whereas
the second order will be called partition-first (or PF for short). As an example, Figs. 1 and 2 show all Cayley trees of size 3 in SF and PF order, respectively.

The orders induced by other combinatorial constructions are similarly inspired. For instance,
in the case of sequences, we induce the corresponding order from the isomorphism

\[ \text{Seq}(A) = \varepsilon + A \times \text{Seq}(A). \tag{3} \]

\(^2\) We are making a slight abuse of notation here: we have refrained from writing \(\alpha = (Z_1, (Z_3, (Z_2, \varepsilon)))\),
\(\beta = (Z_2, (Z_1, \varepsilon))\), etc., in favor of the usual and more readable form \(\alpha = 132, \beta = 21, \ldots\).
The same isomorphism is used for labelled sequences, replacing the unlabelled operators
\((+ \times)\) by their labelled counterparts \((+, ⋆)\). We will give all the orders not presented here
(sets, cycles, powersets, multisets, etc.) in Sections 3 and 4, as we present the corresponding
algorithms.

2.3. Iterators

A commonly encountered abstraction in Computer Science is that of an iterator, which
is an object that can be used to make a traversal in a collection of other objects. In our
case, the collection of objects to be traversed is not explicitly stored anywhere, but has to be
computed on-the-fly as the iterator advances through the collection. That means that in our
context an iterator must contain itself the current combinatorial object in the collection. Our
iterators will also store additional information which will be used to advance the iterator to
the next current object.

Let us consider the following example of use:

```plaintext
FunctGraphSpec := \{F, (F = Set(Cycle(T)), T = Z * Set(T)), labelled\};
itr := init_iter(FunctGraphSpec, size = 20);
while not is_last(itr) do
  DoSomething(get_object(itr));
  itr := next(itr);
endwhile
```

The first line provides a specification `FunctGraphSpec` for functional graphs in three
parts: the class that we want to generate \(F\), the set of equations of the specification,\(^3\) and
an explicit indication that the specification is for a labelled class. The loop applies (exactly
once) the function `DoSomething` to each functional graph of size 20.\(^4\) We assume that if
the iterator `itr` represents the actual last functional graph of size 20 and we call `next` upon

\(^3\) It is not very difficult to implement the algorithms so that they accept a more flexible syntax than that suggested
by our Definition 2.

\(^4\) Of course, you will never apply this particular example in practice. The number of functional graphs of size
20 is so big that even if we spent one nanosecond to generate and process each object it will take more than 35
million centuries to finish the task.
it, then \( \textit{itr} \) will afterwards represent a special fictitious object (this is usually known as past-the-end convention). The Boolean function \( \text{is\_last} \) returns \text{true} when the iterator is the fictitious object and returns \text{false} otherwise.

The most crucial part of our task is the design of the functions \textit{init\_iter} and \textit{next}. The former sets up the appropriate representation and additional information for the first object, and the latter advances from the current object to the next one. There are many ways in which the current combinatorial object can be represented and the convenience of each one of these will typically depend on the application. Translating the internal representation of the current object used within the iterator \( \textit{itr} \) into the most convenient representation is \textit{get\_object}'s task (or of the combination of \textit{DoSomething} and \textit{get\_object}). For the time being, we will assume that \textit{get\_object} produces a linear representation of the current object like

\[
\text{Set}(\text{Cycle}(8 \times \text{Set}(3 \times \text{Set}(1, 9)), 5 \times \text{Set}(7), 2, 6), \text{Cycle}(4))
\]

to represent the functional graph shown in Fig. 3.

But the important aspect here is how the current combinatorial object is internally represented within the iterator: we should look for a suitable internal representation that eases the computation of the next object, while not making it computationally difficult or expensive to “extract” the current object.

We use a tree structure to represent combinatorial objects. The leaves of such structure correspond to atomic objects (of size 1) and to \( \varepsilon \) (of size 0). Internal nodes are “labelled” by the admissible operators, including those that we have seen so far. In addition of the name of the operator, each internal node contains a wealth of information concerning the (sub)object represented by the subtree rooted at that node: its size \( n \), the rank of the object within its class, the class \( \mathcal{A} \) the object belongs to, the number \( a_n \) of objects in \( \mathcal{A}_n \), etc. Notice that some of these fields of information will store large integers and we will need to perform arithmetic operations on them. The nodes might also contain information giving direct access to the linear or conventional representation of the object so that the updates in the tree representation can be readily and efficiently be reflected in the linear representation of the object. We will not describe these improvements in detail, though.
In the case of labelled objects, internal nodes also contain information about the labels used on the object’s atoms. For instance, each node corresponding to a labelled product contains the \((n, j)\)-partition used to relabel the pair, the rank of the partition among the \(\binom{n}{j}\) possible partitions, and some additional data structures which allow for fast update of the atoms’ labels. For example, if we are using \(\text{PF}\) order for labelled objects, in order to move from one object to its successor we should change (if possible) the current partition to the next partition; typically this implies that two labels are exchanged in the partition and thus only a few atoms have to exchange their labels.

The trees that we use to represent objects are binary. When an object belongs to a disjoint union \(A + B\) the root is labelled ‘+’ and only one of its subtrees is used (in particular, we will systematically use the right subtree). The root of the subtree indicates whether the object actually belongs to \(A\) or to \(B\). Internal nodes labelled ‘Seq’, ‘Set’, ‘Cycle’, ... are binary. The reason is that, except for empty sequences, sets, powersets or multisets, we will decompose each object in such classes in two parts. For instance, a labelled cycle in \(\text{Cycle}(A)\) will be decomposed in a first component represented by the left subtree which is some object in the base class \(A\), and a second component represented by the right subtree which is a sequence of \(A\)’s.

Fig. 4 conveys most of the ideas presented so far: the first part of the figure shows the decomposition into cycles of the permutation \(\sigma = 635241\); the binary tree below shows how this particular object is internally represented within an iterator. But notice that if we specify the class of permutations by \(P = \text{Seq}(Z)\) then the internal representation of the same object is totally different.

In the algorithms given in the following sections, we use pseudocode to express them. Given a pointer \(p\) to an internal node (we will say \(p\) is a \(\text{ptrNode}\)), \(p \rightarrow \text{field}\) refers...
to the attribute or field of the given name in the internal node. All internal nodes contain attributes like size (the size of the object), class (the class to which the object belongs), count (the number of objects of that size in the class), rank (the rank of the current object), and several others whose meaning will be explained together with the algorithms. Also, each node contains pointers left and right to the roots of the respective subtrees.

3. Iterating through labelled objects

The generation of labelled objects is conceptually easier and more “homogeneous” than the generation of unlabelled objects, mainly thanks to the concept of boxed products [7] which provides an elegant solution to the problem of iterating through labelled sets and cycles. However, for the generation of labelled objects we will need to find an efficient way to generate the labellings of the objects (we address this issue in Section 3.2), a problem which of course does not arise in the generation of unlabelled objects.

The subsections ahead consider the iteration through disjoint unions, labelled products, sequences, sets and cycles of labelled objects. At the highest level, though, the next routine is nothing but a switch to determine, on the basis of the current object’s class, which one of the routines (e.g., next_union, next_lprod, ...) we need to use. We assume here that next already receives a pointer p to the root node of the current object’s internal representation.

**procedure** next(p: ptrNode) **return** ptrNode
   
   if p → class = ε then ... fi
   if p → class = Atom then ... fi
   if p → class = A + B then return next_union(p) fi
   if p → class = A ⋆ B then return next_lprod(p) fi
   ...

end

3.1. Iterating through labelled disjoint unions

To generate all the objects of size n in A + B is not too difficult (see Algorithm 1). We assume already that the rank r of the current object is strictly smaller than an + bn − 1. If r < an − 1 then the current object is not the last in An and we recursively apply the procedure to obtain the next object in An, in particular, we make a recursive call to function next in the subtree beneath. As we have already said, in the representation of disjoint unions we always use the right subtree and leave the left subtree empty. When r = an − 1 we have to replace the current object (the last one in An) by the first object in Bn using the initialization routine for B. On the other hand, if the current object’s rank is r > an − 1, since r < an + bn − 1 then it means that we have already generated all objects in An and the current object belongs to Bn, so the same idea applies but now to produce the next object in Bn. Of course, the initialization routine for (A + B)n prepares the root node of the iterator and uses An’s initialization routine to get the first object in (A + B)n (unless An = ∅; then we would have to use Bn’s init routine).
Algorithm 1. Computing the next object in a class \( A + B \)

**procedure** next_union\( (p: \text{ptrNode}) \) **return** ptrNode

\[ p \text{ points to a node representing an object in } A + B \]

\[ p \rightarrow \text{rank} := p \rightarrow \text{rank} + 1 \]

**if** \( p \rightarrow \text{rank} \geq p \rightarrow \text{count} \) **then** return \( p \)** fi

**if** \( p \rightarrow \text{right} \rightarrow \text{rank} = p \rightarrow \text{right} \rightarrow \text{count} - 1 \) **then**

\{ \( p \rightarrow \text{right} \) must be the last object in \( A_n \) \}

\[ p \rightarrow \text{right} := \text{init}(\mathcal{B}, p \rightarrow \text{size}) \]

**else**

\{ there are still objects in \( p \rightarrow \text{right} \rightarrow \text{class} \) \}

\[ p \rightarrow \text{right} := \text{next}(p \rightarrow \text{right}) \]

fi

**return** \( p \)

end

In some implementations the replacement of the right subtree representing \( A_n \)’s last object by \( B_n \)’s first object will require that the nodes of the subtree representing the last object in \( A_n \) are explicitly released back to the free storage. In other implementations we may have developed an special purpose memory allocator that recycles nodes. This is particularly useful for leaf nodes, since an object of size \( n \) will exactly contain \( n \) atom nodes and at most \( n + 1 \) \( v \)-nodes. Hence, each time that we need a “new” leaf node we actually reuse one that has just been released. However, in Algorithm 1 and for the rest of this paper, we assume for simplicity that our “programming language” has some garbage collection system that will automatically take care of the unused nodes.

### 3.2. Iterating through labelled products

The algorithm for labelled products is a bit more complicated and depends on the order (SF or PF) that we have chosen. An important component of the algorithm is the method that we use to compute the successor of a given \((n, j)\)-partition, which implicitly imposes the order \( \prec_{S_{n,j}} \).

Let us consider first the generation of labelled products in PF order. For the internal node corresponding to the current object and pointed to by \( p \), the current partition is given by \( p \rightarrow \text{partition} \) and its rank by \( p \rightarrow \text{partition\_rank} \). If this rank is less than \( \binom{n}{j} - 1 \), then we move onto the next labelled object by changing the \((n, j)\)-partition. This is accomplished by the procedure next_label, which besides changing the current partition by the next one, updates a few additional fields to reflect the change in the labels of the atoms. For instance, we can use Nijenhuis–Wilf’s algorithm [16] or Kemp’s algorithm [9] to compute the next \( j \)-subset, as the basis for our algorithm to generate \((n, j)\)-partitions. One important property of the two mentioned algorithms is that they generate all \( j \)-subsets in \( \Theta(\binom{n}{j}) \) time; in other words, both algorithms work in constant amortized time. This property extends to the algorithm that we use in next_label. We will discuss this procedure in more detail later.
Assume now that the rank of the current partition is \( \binom{n}{j} - 1 \). That means, that we have to change at least one of the objects in the pair; if the second component of the pair, represented by the subtree \( p \to \text{right} \) is not the last object in \( B_{n-j} \) then we recursively apply the procedure to the right subtree in order to compute the next object in \( B_{n-j} \). On the other hand, if the object to the right is the last one in its class, we should compute the successor of the object given by \( p \to \text{left} \) in \( A_j \), and change the second component to the first object in \( B_{n-j} \). But if the first component of the current pair is indeed the last object in \( A_j \), we should find the first \( j' > j \) such that \( A_{j'} \cdot B_{n-j'} \neq \emptyset \) and initialize the current object with subtrees representing the first object in \( A_{j'} \) and \( B_{n-j'} \), respectively. Notice the explicit use of the function \( \text{count} \) to find the first \( j' > j \) such that \( a_j \cdot b_{n-j'} \neq 0 \).

A concise description of this algorithm to generate labelled products is given in Algorithm 2.

**Algorithm 2. Computing the next object in a class \( A \cdot B \) in PF order**

```
procedure next_lprod(p: ptrNode) return ptrNode
    \{p points to a node representing an object in \( A \cdot B \)\}
    j := p \to \text{left} \to \text{size}; n := p \to \text{size};
    A := p \to \text{left} \to \text{class}; B := p \to \text{right} \to \text{class}
    p \to \text{rank} := p \to \text{rank} + 1
    if p \to \text{rank} \geq p \to \text{count} then return p fi
    if p \to \text{partition\_rank} < \binom{n}{j} - 1 then
        next_label(p)
        p \to \text{partition\_rank} := p \to \text{partition\_rank} + 1
        return p
    fi
    p \to \text{partition} := \{[1, 2, \ldots, j], \{j + 1, \ldots, n\}\}
    p \to \text{partition\_rank} := 0
    if p \to \text{right} \to \text{rank} < p \to \text{right} \to \text{count} - 1 then
        p \to \text{right} := \text{next}(p \to \text{right}); return p
    fi
    if p \to \text{left} \to \text{rank} < p \to \text{left} \to \text{count} - 1 then
        p \to \text{left} := \text{next}(p \to \text{left})
        p \to \text{right} := \text{init}(B, n - j)
        return p
    fi
    j := j + 1
    while \( j \leq n \) and (\( \text{count}(A, j) = 0 \) or \( \text{count}(B, n - j) = 0 \)) do
        j := j + 1
    end
    p \to \text{partition} := \{[1, 2, \ldots, j], \{j + 1, \ldots, n\}\}
    p \to \text{left} := \text{init}(A, j)
    p \to \text{right} := \text{init}(B, n - j)
    return p
end
```
We complete our description of the generation of labelled products by giving now the missing
details of the procedure \texttt{next\_label}. The root node of the current object belonging to \(A_j \ast B_{n-j}\) stores the current partition as an array of size \(n\) called \texttt{partition}. The contents of
the array are stored in ascending order. We will also need an additional array \texttt{inv\_part} such
that if \(\texttt{partition}[i] = k\) then \(\texttt{inv\_part}[k] = i\), for \(1 \leq i, k \leq n\). The node will also contain
the permutation of the labels attached to the object’s atoms in an array \texttt{perm} of size \(n\) and its
inverse in another array \texttt{inv\_perm}. Our routine \texttt{next\_label} is based upon Kemp’s algo-
rithm for the next \(j\)-subset. Nijenhuis–Wilf’s algorithm could have been similarly adapted.
One of the modifications that we need is to update the \texttt{perm} and \texttt{inv\_perm} arrays to reflect
the changes made in \texttt{partition}. Basically, each time the algorithm swaps a label \(\ell\) with a
label \(\ell’\) in \texttt{partition} then we have to perform the following changes:

\[
p := \texttt{inv\_perm}[\ell]; q := \texttt{inv\_perm}[\ell’]\]
\[
\texttt{perm}[p] \leftrightarrow \texttt{perm}[q]\]
\[
\texttt{inv\_perm}[\ell] \leftrightarrow \texttt{inv\_perm}[\ell’]
\]

But other modifications are necessary to be able to iterate through full \((n, j)\)-partitions.
Both Kemp’s and Nijenhuis–Wilf’s algorithms maintain only the \(j\)-subset of the current
partition (the \((n-j)\)-subset in the partition is implicit), and they do it only for \(j \leq \lfloor n/2 \rfloor\).
Thus we must modify Kemp’s algorithm in order to maintain the full \((n, j)\)-partition for
\(0 \leq j \leq n\), while still guaranteeing that it works in constant amortized time. The situation
where \(j > \lfloor n/2 \rfloor\) can be dealt with reversing the rôles played by the first half and the second
half of the partition array (and keeping track of this fact, to maintain the lexicographic order
of the partitions).

But maintaining the full \((n, j)\)-partition and being able to compute the next one in constant
amortized time is more complicated; the array \texttt{inv\_part} is a must here, because it allows
us to know where and how do we have to update a label which “moves” from the \(j\)-subset
to the \((n-j)\)-subset or vice-versa.

The bulk of \texttt{next\_label} code is given in Algorithm 3. For simplicity, we have not
included the code necessary to update \texttt{perm} and \texttt{inv\_perm} arrays; we have also omitted
references to the node that stores the corresponding arrays.

The cost to generate the \(\binom{n}{j}\) partitions is proportional to \(\binom{n}{j}\); hence, generating each
partition takes constant amortized time. The only difference in performance between Kemp’s
algorithm and Algorithm 3 comes from lines (22) to (25). A simple counting argument
reveals that there are exactly \(\binom{n-r-2}{n-r-j}\) \((n, j)\)-partitions for which the loop (22)–(25) (or its
analogue for \(j > \lfloor n/2 \rfloor\)) makes exactly \(r\) iterations. Hence, the total cost to generate all
\(\binom{n}{j}\) partitions contributed by these lines is proportional to

\[
\sum_{r=2}^{n-j-1} r \binom{n-r-2}{n-r-j} = \frac{n+j-2}{n-j-2} \binom{n-3}{j} - (n-j)
\]

which is \(\mathcal{O}\left(\binom{n}{j}\right)\).
Algorithm 3. Computing the next \((n, j)\)-partition in next_label

1. \textbf{if} \(j > \lfloor n/2 \rfloor\) \textbf{then} \ldots
2. \textbf{else} \(i := j\)
3. \textbf{while} \(i \geq 1\) \textbf{and} \(\text{partition}[i] = n - j + i\) \textbf{do}
4. \hspace{1em} \(i := i - 1\)
5. \textbf{end}
6. \(i' := \text{partition}[i] + 1; i'' := \text{partition}[i]\)
7. \(\text{partition}[i] \leftrightarrow \text{partition}[^{\text{inv_part}}[i']]\)
8. \(\text{inv_part}[i''] \leftrightarrow \text{inv_part}[i']\)
9. \{Update also here \(\text{perm}\) and \(\text{inv_perm}\)\}
10. \textbf{if} \(\text{partition}[i] + 1 \neq \text{partition}[i + 1] - 1\) \textbf{then}
11. \hspace{1em} \textbf{for} \(k := i + 1\) \textbf{to} \(j\) \textbf{do}
12. \hspace{2em} \(k' := \text{partition}[k - 1] + 1; k'' := \text{partition}[k]\)
13. \hspace{2em} \(\text{partition}[k] \leftrightarrow \text{partition}[^{\text{inv_part}}[k']]\)
14. \hspace{2em} \(\text{inv_part}[k''] \leftrightarrow \text{inv_part}[k']\)
15. \hspace{1em} \textbf{if} \(\text{inv_part}[k''] < n\) \textbf{and}
16. \hspace{2em} \(\text{partition}[^{\text{inv_part}}[k'']] > \text{partition}[^{\text{inv_part}}[k''] + 1]\) \textbf{then}
17. \hspace{3em} \(z := \text{inv_part}[k'']\)
18. \hspace{3em} \textbf{for} \(t := k + 1\) \textbf{to} \(j\) \textbf{do}
19. \hspace{4em} \(\text{partition}[t] := \text{partition}[t - 1] + 1\)
20. \hspace{4em} \(\text{inv_part}[\text{partition}[t]] := t\)
21. \hspace{3em} \textbf{end}
22. \hspace{2em} \textbf{end}
23. \hspace{2em} \textbf{for} \(t := z\) \textbf{to} \(n\) \textbf{do}
24. \hspace{3em} \(\text{partition}[t] := t\)
25. \hspace{3em} \(\text{inv_part}[t] := t\)
26. \hspace{2em} \textbf{end}
27. \hspace{1em} \textbf{break} \{finishes loop on \(k\) (lines 11 to 28)\}
28. \textbf{fi}
29. \textbf{end}
30. \textbf{fi}

The generation of labelled products in SF order is quite similar to the generation in PF order, but we do not change the current object’s partition until all pairs \((\alpha, \beta)\) have been generated and the current partition implicitly applied to all them. Algorithm 4 is just an outline, where we have tried to emphasize the (small) differences with Algorithm 2.

For both SF and PF order, the initialization for \(A \bowtie B\) is achieved by finding the smallest \(j \geq 0\) such that \(A_j \bowtie B_{n-j} \neq \emptyset\) and setting up a root node with the initial partition,
partition\_rank = 0, rank = 0, etc. Finally, its subtrees are initialized with the first objects in \( A_j \) and \( B_{n-j} \), respectively.

---

Algorithm 4. Computing the next object in a class \( A \star B \) in SF order

**procedure** next\_lprod\((p:\) ptrNode) **return** ptrNode

\( \{p \) points to a node representing an object in \( A \star B \}\)

\( \ldots \)

if \( p \rightarrow right \rightarrow rank < p \rightarrow right \rightarrow count - 1 \) then

\( p \rightarrow right := \text{next}(p \rightarrow right); \text{return} \ p \)

fi

if \( p \rightarrow left \rightarrow rank < p \rightarrow left \rightarrow count - 1 \) then

\( p \rightarrow left := \text{next}(p \rightarrow left) \\
\quad p \rightarrow right := \text{init}(B, n - j) \)

**return** \( p \)

fi

if \( p \rightarrow partition\_rank < \binom{n}{j} - 1 \) then

\( \text{next\_label}(p) \\
\quad p \rightarrow partition\_rank := p \rightarrow partition\_rank + 1 \)

**return** \( p \)

fi

\( j := j + 1 \)

while \ldots \ do

\( j := j + 1 \)

end

\( \ldots \)

end

---

3.3. Iterating through sequences, sets and cycles

Sequence objects are represented by a single leaf corresponding to \( \varepsilon \) or a binary tree whose left subtree represents the head of the sequence and whose right subtree represents the remaining sequence, the tail of the sequence. The order for \( \text{Seq}(A) \) and the algorithm to generate its objects directly stem from the isomorphism

\[
\text{Seq}(A) = \varepsilon + A \star \text{Seq}(A).
\]

(4)

The reader can readily convince herself that next\_lprod works just fine to generate sequences. Also, to initialize a sequence we use similar ideas as before.

As we pointed out at the beginning of this section, the generation of labelled sets and cycles is greatly simplified thanks to the isomorphisms they satisfy in terms of the so called boxed product. Boxed products, denoted \( A \boxtimes \), are very similar to labelled products, with the condition that each pair in \( \alpha \boxtimes \beta \) must be relabelled in such a way that the smallest label is attached to some atom in the first component. Since the smallest label must be used to label some \( \alpha \)-atom there are only \( \binom{n-1}{j-1} \) objects in \( \alpha \boxtimes \beta \), where \( j = |\alpha| \) and \( n = |\alpha| + |\beta| \).
The generalization of the SF and PF orders to boxed products is immediate; we should only remember that the \((n, j)\)-partitions used to relabel the pairs \((z, \bar{z})\) are restricted to contain the smallest label in its first subset and so there are only \(\binom{n-1}{j-1}\) of them.

Once boxed product and the order(s) it induces have been introduced, we will use the following isomorphisms in order to generate labelled sets (‘Set’) and labelled cycles (‘Cycle’):

\[
\text{Set}(A) = \varepsilon + A \Box \ast \text{Set}(A),
\]

\[
\text{Cycle}(A) = A \Box \ast \text{Seq}(A).
\]

Notice the similarity of (5) with (4). We represent a labelled set as a single leaf if the set is empty; otherwise, we have a binary tree where the left subtree represents the component of the base class with the smallest label (the leader), and the right subtree represents a set with the remaining components of the set. In other words, the “canonical” form of a set \(S = \{x_1, x_2, \ldots, x_k\}\) with \(x_i \in A\) is that in which the \(x_i\)'s have been arranged in ascending order of their respective smallest labels.

To represent a cycle, we set up a binary tree whose left subtree represents the leader; once we have “open” the cycle at the leader, the rest is a sequence of objects in the base class. Again the criterion to choose the leader in a cycle is the same as for sets: the component that contains the smallest label among its atoms.

From the discussion above, we only need an algorithm \texttt{next_bprod} for boxed products to cope with both Sets and Cycles. Furthermore, \texttt{next_bprod} is almost identical to \texttt{next_lprod}, except for the way it handles the partitions which relabel the atoms (see Algorithm 5). Since the procedure \texttt{next_label} generates them in lexicographic order, the first \(\binom{n-1}{j-1}\) partitions thus obtained are exactly those that contain the smallest label in its first subset, so we can still use the same procedure. In order to initialize a set or a cycle we proceed much in the same manner as for sequences.

---

**Algorithm 5. Computing the next object in a class \(A \Box \ast B\) in PF order**

```plaintext
procedure next_lprod(p: ptrNode) return ptrNode
    \{p points to a node representing an object in \(A \Box \ast B\)\}
    \{Almost all code is identical to that of Algorithm 2\}
    j := p -> left -> size; n := p -> size

    ... if p -> partition_rank < \(\binom{n-1}{j-1} - 1\) then
        next_label(p)
        p -> partition_rank := p -> partition_rank + 1
    return p
    fi

    ...
```
4. Iterating through unlabelled objects

The generation of unlabelled disjoint unions, unlabelled products and unlabelled sequences poses no special difficulty. The algorithms to cope with these operators are actually easier than the corresponding labelled algorithms in Section 3 since they do not have to update labels, partitions, etc. In addition, we only have to consider one order for products, namely, the lexicographic order.

The generation of unlabelled multisets and powersets is based on the decomposition of objects into blocks of components of the same size, with the leading block being the one containing the components of smallest size. If the smallest component in two given sets or powersets is of the same size then the smaller set is that with the greater number of components of that size, i.e., whose leading block has the greater number of components. When two sets or powersets of A's have leading blocks with components of the same size, say j, and the same number of components, then the components within the block are considered according to the order in A and compared lexicographically.

Let \( C = \text{MSet}(A) \). The order in \( C_n \), when \( n > 0 \), is induced by the isomorphism

\[
C_n = \bigcup_{j=1}^{n} \bigcup_{r=\lfloor n/j \rfloor}^{1} \text{MSet}(A_j, \text{card} = r) \times \text{MSet}(A_{>j})_{n-jr}, \quad n > 0,
\]

where \( \text{MSet}(A_j, \text{card} = r) \) is the class of multisets with exactly \( r \) components which are objects in \( A_j \), and \( \text{MSet}(A_{>j}) \) is the class of multisets whose components are objects in \( A \) of size strictly larger than \( j \). To complete the definition of the order \( \prec_{C_n} \), we need to define the order in \( \text{MSet}(A_j, \text{card} = r) \); given two objects \( \beta \) and \( \beta' \) in that class, \( \beta < \beta' \) if the sorted sequence of the ranks (with repetitions) of the components of \( \beta \) is lexicographically smaller than the sorted sequence of the ranks of the components of \( \beta' \).

For powersets, the definition of the order is basically the same. In particular, if \( C = \text{PSet}(A) \) then the order in \( C_n \) is induced by the isomorphism

\[
C_n = \bigcup_{j=1}^{n} \bigcup_{r=\lfloor n/j \rfloor}^{1} \text{PSet}(A_j, \text{card} = r) \times \text{PSet}(A_{>j})_{n-jr}, \quad n > 0,
\]

together with the definition of order in \( \text{PSet}(A_j, \text{card} = r) \); given two objects \( \beta \) and \( \beta' \) in that class, \( \beta < \beta' \) if the sorted sequence of the ranks of the components of \( \beta \) is lexicographically smaller than the sorted sequence of the ranks of the components of \( \beta' \).

The only remaining difficulty is to generate multisets and powersets of a fixed cardinality and components of a given size, that is, how to generate the leading blocks of multisets and powersets. From there, we only need a slight modification of the algorithm for products in order to generate multisets and powersets.

We also need to represent multisets and powersets in a convenient way. Following the isomorphisms (7) and (8), a non-empty object \( \gamma \) is seen as a pair \((B, \gamma')\) with \( B \) a block of \( r \) components of size \( j \) in ascending order and \( \gamma' \) a multiset or powerset with components of size \( > j \). The root node is labelled \( \text{MSet} \) or \( \text{PSet} \) and contains the usual fields of information; additionally it contains a field \( \text{min\_size} \) that stores the minimal possible size among its components. Thus the structure of the tree and its right child is the same.
The representation of blocks is different since it must represent a collection of exactly \( r \) objects of size \( j \). The root node of such a block is labelled ‘mblock’ (when repetitions are allowed) or ‘pblock’ (when no repetitions are allowed), and it contains the number of objects or cardinality of the block. The left subtree of a mblock of \( r \) objects contains a \( \lambda \) object, that is, the representation of some object \( x \) in \( A_j \) (in particular, the smallest object in the mblock according to \( \prec_{A_j} \)) together with the number of times, say \( d \), that it is present in the mblock; and the right subtree represents a mblock of \( r - d \) objects. On the other hand, the left subtree of a pblock of \( r \) objects represents the smallest object in the pblock according to \( \prec_{A_j} \), and the right subtree represents a pblock of \( r - 1 \) objects. Both mblock and pblock nodes contain an attribute \( \text{min\_rank} \) which corresponds to the smallest possible rank for an object within the block. This attribute is 0 for the first mblock or pblock node in the left subtree of a \( \text{MSet} \) or \( \text{PSet} \) node, otherwise it is one more than the rank of the (possibly repeated) object represented in a left subtree which is a sibling of the mblock or pblock node.

Fig. 5 schematically depicts the internal representation of the powerset

\[ \gamma = \{aa, ba, aba, abb, bbb, abaa, bbba, bbbaab\} \in \text{PSet}(\text{Seq}(a + b)). \]

‘PSet’ nodes are represented by oval nodes, together with the values of the \( \text{size} \) and \( \text{min\_size} \) attributes. Pblock nodes are represented by circular nodes with thick border; within each pblock node we write the values of the \( \text{size} \) and \( \text{min\_rank} \) attributes. The components of the powerset belong to \( \text{Seq}(a + b) \) but we have shown them “collapsed” into single rectangular nodes.

For instance, since the leading block of \( \gamma \) contains two components of size 2, the root node of the right subtree has \( \text{min\_size} = 3 \), as no component of that powerset can be of size smaller than 3.

Fig. 6 corresponds to the internal representation of the multiset

\[ \{aa, aa, ba, aba, aba, abb, bbb, abaa, bbba, bbbaab\} \in \text{MSet}(\text{Seq}(a + b)). \]

We use the same set of conventions as in Fig. 5; additionally, the diamond-shaped nodes represent ‘\( \lambda \)’ nodes, and store the total size (i.e., the size of the component times its frequency) and the number of times the component occurs in the multiset.

The algorithms to iterate through unlabelled powersets and multisets are very similar and they rely on the algorithms next_pblock and next_mblock, respectively. We give in Algorithm 6 the pseudocode to generate powersets; except for minor details, the code for
multisets is identical. Notice that next_pset uses the procedure init_pset, which receives a specification of the class \( \mathcal{A} \) of the components, the total size of the powerset, and the minimum possible size for the components of the powerset. The procedure init_pset initializes the class attribute of the root node to \( \text{PSet}(\mathcal{A}) \) which is not enough, because we have an additional restriction on the size of the components; hence, the special attribute min_size. There is also an analogous procedure init_mset to initialize multisets, with the same three parameters, and similar considerations apply. These procedures use the procedures init_pblock and init_mblock, respectively, each receiving four parameters: a specification for the class \( \mathcal{A} \) of the components, the total size of the block, the size of the components of the block, and the smallest allowed rank among its components. The implementation of init_mblock and init_pblock relies upon efficient algorithms for unranking in \( \mathcal{A}_j \); we use algorithms such as those developed in [13]. Since the class attribute of a mblock or a pblock is never checked, we do not have to initialize it; if necessary, it could be initialized with something like \( \text{PSet}(\mathcal{A}, \text{card} = r) \), as we are not able to express the additional condition that all components are of the same size, say \( j \).

Also, both next_pset and init_pset need the specialized counting routine \( \text{count\_PowerSet}(\mathcal{A}, n, j) \). This routine uses the generic counting function count in order to compute the number of powersets of \( \mathcal{A} \)'s of size \( \geq j \) with total size \( n \). For the generation of multisets, there exists a similar specialized procedure \( \text{count\_MultiSet}(\mathcal{A}, n, j) \) (Fig. 6).

Algorithm 8 gives the pseudocode for the generation of mblocks, the pseudocode for pblocks being very similar and in fact simpler. As usual, when computing the next mblock we try to change the right subtree first. If it is not possible, we decrease by one the number of occurrences of the smallest component of the mblock (the object in \( \mathcal{A}_j \) of smallest rank) and start with the smallest possible mblock in the right. If the number of occurrences of the smallest component is already one, we have to move to the next object in \( \mathcal{A}_j \), with maximum number occurrences and initialize the right subtree to an empty mblock.

---

Algorithm 6. Computing the next object in a class \( \text{PSet}(\mathcal{A}) \)

**procedure** next_pset\( (p: \text{ptrNode}) \) \textbf{return} ptrNode

\( \{ p \) points to a node representing an object in \( \text{PSet}(\mathcal{A}) \}\)

\( p \rightarrow \text{rank} := p \rightarrow \text{rank} + 1 \)
if $p \rightarrow \text{rank} \geq p \rightarrow \text{count}$ then return $p$
fi

$k := p \rightarrow \text{left} \rightarrow \text{size}; n := p \rightarrow \text{size}$
$A := p \rightarrow \text{left} \rightarrow \text{class}$

if $p \rightarrow \text{right} \rightarrow \text{rank} < p \rightarrow \text{right} \rightarrow \text{count} - 1$ then

$p \rightarrow \text{right} := \text{next}_\text{pset}(p \rightarrow \text{right}); \text{return } p$
fi

if $p \rightarrow \text{left} \rightarrow \text{rank} < p \rightarrow \text{left} \rightarrow \text{count} - 1$ then

$p \rightarrow \text{left} := \text{next}_\text{pblock}(p \rightarrow \text{left})$
$k := p \rightarrow \text{left} \rightarrow \text{size}; j := p \rightarrow \text{left} \rightarrow \text{left} \rightarrow \text{size}$
$p \rightarrow \text{right} := \text{init}_\text{pset}(A, n - k, j + 1)$
\text{return } p
fi

{Computes the next powerset by changing the number of components $r = k/j$ or their size $j$ in the pblock.}
{See Algorithm 7}
\text{return } \text{change}_\text{pblock}(p)$
end

Algorithm 7. Computing the next object in a class $\text{PSet}(A)$ (II)

\textbf{procedure} \text{change}_\text{pblock}(p: \text{ptrNode}) \textbf{return} \text{ptrNode}

\begin{align*}
&j := p \rightarrow \text{left} \rightarrow \text{left} \rightarrow \text{size}; r := k/j - 1 \\
&A := p \rightarrow \text{left} \rightarrow \text{left} \rightarrow \text{class}
\end{align*}

\text{while } r > 0 \text{ and } \text{count}_\text{PowerSet}(A, n - j \cdot r, j) = 0 \text{ do}

\begin{align*}
&r := r - 1 \\
&\text{end}
\end{align*}

if $r > 0$ then

\begin{align*}
&p \rightarrow \text{left} := \text{init}_\text{pblock}(A, j \cdot r, j, 0) \\
&p \rightarrow \text{right} := \text{init}_\text{pset}(A, n - j \cdot r, j + 1)
\end{align*}
\text{return } p
fi

\begin{align*}
&j := j + 1; r := \lfloor n/j \rfloor \\
&\text{while } j \leq n \text{ and } (\text{count}(A, j) = 0 \text{ or } \text{count}_\text{PowerSet}(A, n - j \cdot r, j) = 0) \text{ do}
\end{align*}

if $\text{count}(A, j) \neq 0$ then

\begin{align*}
&\text{while } r > 0 \text{ and } \text{count}_\text{PowerSet}(A, n - j \cdot r, j) = 0 \text{ do}
\end{align*}

\begin{align*}
&r := r - 1 \\
&\text{end}
\end{align*}

fi

if $\text{count}(A, j) = 0$ \text{ or } $r = 0$ then $j := j + 1; r := \lfloor n/j \rfloor$ fi

\begin{align*}
&p \rightarrow \text{left} := \text{init}_\text{pblock}(A, j \cdot r, j, 0) \\
&p \rightarrow \text{right} := \text{init}_\text{pset}(A, n - j \cdot r, j + 1)
\end{align*}
\text{return } p
end
Algorithm 8. Computing the next mblock

```plaintext
procedure next_mblock(p: ptrNode) return ptrNode
    { p points to a node representing an object in MSet(A_j, card = r) }
    p → rank := p → rank + 1
    if p → rank ≥ p → count then return p fi
    if p → right → rank < p → right → count - 1 then
        p → right := next_mblock(p → right)
        return p
    fi
    if p → left → occurrences > 1 then
        j := p → left → left → size
        rnk := p → left → left → rank
        p → left → occurrences := p → left → occurrences - 1
        p → left → size := p → left → size - j
        A := p → left → left → class
        p → right := init_mblock(A, p → size - p → left → size, j, rnk + 1)
        return p
    fi
    { p → left → left → rank < p → left → left → count - 1 }
    j := p → left → left → size
    rnk := p → left → left → rank
    p → left → occurrences := p → size / j
    p → left → size := p → size
    p → left → left := next(p → left → left)
    p → right := e
    return p
end
```

Last but not least, unlabelled cycles do not admit a nice recursive decomposition so we need a rather different approach to iterate through them. Since the algorithm for cycles significantly differs from the other algorithms in this paper and it is not amenable to the same type of analysis, we only describe it very briefly here. It combines Sawada’s algorithm for k-ary cycles of fixed content [19] and the algorithm for multisets which we have described in this section. Even though unlabelled cycles are dealt with in a different way, the representation of cycles and the algorithm have been designed so that they can be integrated and seamlessly work together with the other algorithms. Also, the algorithm works in constant amortized time per generated cycle (provided that the components could be generated in constant amortized time, which indeed is the case, see Section 5). For the complete details, see [15].

5. The performance

We have anticipated already that all the algorithms in our framework work in constant amortized time. In other words, if \( \Lambda A_n \) denotes the cost of generating all the objects of size \( n \) in a class \( A \) and \( a_n \) is the number of generated objects then \( \Lambda A_n = \Theta(a_n) \).
We do not take into account the cost of calling \texttt{count} at several places of the algorithms of the previous sections. All the necessary \texttt{count}'s can be computed only once and conveniently stored into tables for later use, either on-demand or in a preprocessing phase. The cost contributed by the computation of counts is \( O(\min\{n^2, \sum_{0 \leq k \leq n} a_k\}) \) and can be safely disregarded.

Also, we will charge an unitary cost to all non-recursive steps made in each of the algorithms of the previous section. This is clearly an oversimplification; for instance, generating the next object in \( \mathcal{A} + \mathcal{B} \) requires less non-recursive overhead than generating the next object in \( \mathcal{A} \times \mathcal{B} \). In a more realistic setting we should charge each operator the cost of the non-recursive steps to generate an object: to generate one object in \( \mathcal{A} + \mathcal{B} \) this would be \( c_+ \), for products it would be \( c_x \), and so on, for suitable constants \( c_+, c_x, \ldots \). But the fact that the algorithms' performance is CAT is not affected by this simplification, hence we proceed with this simple accounting scheme.

Another assumption that we make in our analysis is that the leaf nodes are generated at no cost. There are only to different such nodes: \( \varepsilon \) and atoms and we do not have to perform any arithmetic computations to construct them.

All that said, let \( \mathcal{A} \) be some unlabelled class. The cumulating generating function \( \mathcal{A}(z) \) is

\[
\sum_{n \geq 0} \mathcal{A}_n z^n = \sum_{x \in \mathcal{A}} c(x) z^{|x|},
\]

(9)

where \( c(x) \) is the cost of generating \( x \) from its predecessor (or initializing \( x \) if it is the first object in \( \mathcal{A}_{|x|} \)). Similarly, for a labelled class \( \mathcal{A} \), the cumulating generating function is given by

\[
\sum_{n \geq 0} \mathcal{A}_n z^n / n! = \sum_{x \in \mathcal{A}} c(x) z^{|x|} / |x|!.
\]

(10)

Once we have introduced cumulating generating functions, the main result of this section can be expressed as \( \mathcal{A}_n = [z^n] \mathcal{A}(z) = \Theta([z^n] \mathcal{A}(z)) \) for any admissible unlabelled class \( \mathcal{A} \), and \( \mathcal{A}_n = n! \cdot [z^n] \mathcal{A}(z) = \Theta(n! \cdot [z^n] \mathcal{A}(z)) \) for any admissible labelled class \( \mathcal{B} \).

Before going on, we need a few additional definitions. First, for any predicate \( P \), we define \( \| P \| = 1 \) if \( P \) is true, and \( \| P \| = 0 \) if \( P \) is false. Also, for any unlabelled class \( \mathcal{C} \), we define

\[
\| \mathcal{C} \| = \sum_{n \geq 0} \| \mathcal{C}_n \neq 0 \| z^n,
\]

whereas for labelled classes the corresponding definition is

\[
\| \mathcal{C} \| = \sum_{n \geq 0} \| \mathcal{C}_n \neq 0 \| z^n / n!.
\]

For instance, if an unlabelled class \( \mathcal{A} \) contains objects of all sizes then \( \| \mathcal{A} \| = 1/(1 - z) \).
5.1. The cost of generating unlabelled classes

We consider first the performance of the generation of admissible unlabelled classes. Recall that we assume that leaf nodes can be constructed at no cost, hence, \( A\emptyset = A\epsilon = AZ = 0 \).

For disjoint unions, the cumulated cost is basically the sum of cumulated costs, but we must take care to charge the cost corresponding to computing the next of the last element in \( A \) if \( A \neq \emptyset \), and no cost otherwise; this is accounted for by the terms \( \|A\| + \|B\| - \|A + B\| \).

Hence,

\[
A(A + B) = AA + AB + \|A\| + \|B\| - \|A + B\|.
\]  

(11)

The cost of generating sequences follows from (11) and (12), since

\[ A(\text{Seq}(A)) = A\epsilon + A(\text{A} \times \text{Seq}(A)) + \|\epsilon\| + \|\text{A} \times \text{Seq}(A)\| - \|\text{Seq}(A)\| \]

\[ = AA\text{Seq}(A) + A\|\text{Seq}(A)\| + \|A\|\|\text{Seq}(A)\| + 1 - \|\text{Seq}(A)\|. \]

Isolating \( AA \) yields

\[
A(\text{Seq}(A)) = \frac{(AA + \|A\| - 1)\|\text{Seq}(A)\| + 1}{1 - A}.
\]  

(13)
For multisets and power sets, the corresponding rules can be derived from the rules for sums and products, but they do not take a nice form

\[
\begin{align*}
\text{MSet}(A)^n &= \left( \sum_{j=1}^{n} \sum_{r=1}^{\lfloor n/j \rfloor} \left( \frac{a_j + r - 1}{r} \right) \text{MSet}(A_{>j})^{n-jr} \right) \\
&+ \text{MSet}(A_j, \text{card} = r) \left[ \text{MSet}(A_{>j})^{n-\lfloor n/j \rfloor} \right] \\
&+ \left[ \text{MSet}(A_j, \text{card} = r) \neq \emptyset \right] \left[ \text{MSet}(A_{>j})^{n-\lfloor n/j \rfloor} \neq \emptyset \right] \\
&- \left[ \text{MSet}(A) \neq \emptyset \right] \\

de (14)
\end{align*}
\]

and

\[
\begin{align*}
\text{PSet}(A)^n &= \left( \sum_{j=1}^{n} \sum_{r=1}^{\lfloor n/j \rfloor} \left( \frac{a_j r}{r} \right) \text{PSet}(A_{>j})^{n-\lfloor n/j \rfloor} \right) \\
&+ \text{PSet}(A_j, \text{card} = r) \left[ \text{PSet}(A_{>j})^{n-\lfloor n/j \rfloor} \right] \\
&+ \left[ \text{PSet}(A_j, \text{card} = r) \neq \emptyset \right] \left[ \text{PSet}(A_{>j})^{n-\lfloor n/j \rfloor} \neq \emptyset \right] \\
&- \left[ \text{PSet}(A) \neq \emptyset \right] \\

de (15)
\end{align*}
\]

The analysis of the cost of these combinatorial constructions can be further carried on assuming \( \text{MSet}(A_{>j})^n \approx \text{MSet}(A)^n \) and \( \text{PSet}(A_{>j})^n \approx \text{PSet}(A)^n \) (the right-hand sides are in fact upper bounds); we also need to compute the cost of generating mblocks and pblocks, that is, \( A(\text{MSet}(A_j, \text{card} = r)) \) and \( A(\text{PSet}(A_j, \text{card} = r)) \). Notice that we have to iterate through mblocks and pblocks of size \( jr \), since they have a fixed number \( r \) of components which are objects of size \( j \) in some class \( A \); therefore, the cumulating generating functions have only one term corresponding to \( z^{jr} \).

In particular, each object in \( A_j \) is generated exactly once to act as the leader the mblock, and we have the terms for the costs of generating mblocks with components of larger rank when the leader occurs once, twice, three times, etc.:

\[
\text{MSet}(A_j, \text{card} = r) = A_j A_j + \sum_{1 \leq i < r} \sum_{x \in A_j} \text{MSet}(A_j^{(x)}, \text{card} = i),
\]

where \( A_j^{(x)} \) is the subset of objects of size \( j \) in \( A \) preceded by \( x \) (i.e., whose rank is larger than the rank of \( x \)).

The rule for pblocks is similar since each object in \( A_j \) is generated exactly once to act as the leader the pbblock and the rest accounts for the cost of generating the pblock with \( r - 1 \) components:

\[
\text{PSet}(A_j, \text{card} = r) = A_j A_j + \sum_{x \in A_j} \text{PSet}(A_j^{(x)}, \text{card} = r - 1),
\]

**Theorem 1.** For any unlabelled admissible class \( A \),

\[
\mu_{A, n} = \frac{A_A n}{a_n} = \Theta(1).
\]
Proof. The statement of the theorem follows from the fact that
\[ A\mathcal{A}(z) = A(z) - \mathcal{A}(z) \]  
(16)
for any unlabelled admissible class \( A \) which does not involve the operator ‘Cycle’ in its
specification, together with the specific proof that we have developed for classes involving
unlabelled cycles in [15].

Eq. (16) is trivially true for \( A = \emptyset \), \( A = e \) and \( A = Z \). For the other operators, we can
establish the results by structural induction. For instance, if \( C = A + B \) then
\[ A\mathcal{C} = A\mathcal{A} + A\mathcal{B} + A[\mathcal{A}] + A[\mathcal{B}] - [A + B] = A + B - [A + B] = C - [C]. \]
For a product \( C = A \times B \), we get
\[ A\mathcal{C} = A\mathcal{A}B + A[\mathcal{A}]B + A[\mathcal{B}] - [A \times B] = A(B - [B]) + (A - [A])[B] + A[B] - [A \times B] = AB - [A \times B] = C - [C]. \]
Since sequences are defined in terms of sums and products the result holds, but we can also
check it directly by applying structural induction on (13), so for \( C = \text{Seq}(A) \),
\[ A\mathcal{C} = \frac{(A\mathcal{A} + [\mathcal{A}] - 1)[\mathcal{C}] + 1}{1 - A} = \frac{(A - 1)[\mathcal{C}] + 1}{1 - A} = C - [C]. \]
as \( C(z) = 1/(1 - A(z)) \).

For multisets, using structural induction we have
\[ A\text{MSet}(A_{>j}) = \text{MSet}(A_{>j})(z) - [\text{MSet}(A_{>j})], \]
where we have used \( \text{MSet}(A_{>j})(z) \) to denote the counting generating function of \( \text{MSet}(A_{>j}) \). On the other hand, assuming that the statement of the theorem holds for \( A\text{MSet}(A_{(z)}, \text{card} = i) \), \( 1 \leq i < k \), we can conclude that
\[ A\text{MSet}(A_{j}, \text{card} = r) = \binom{a_j + r - 1}{r} z^r - [\text{MSet}(A_{j}, \text{card} = r)]. \]
(18)
Actually, the result above is only almost true because the leading object of a mblock occurring \( d \) times in the mblock has not to be generated \( d \) times but only once. However, assuming that (18) is true does not harm as it would be an overestimation of the real cost.

Substituting (17) and (18) into (14) yields
\[ A\text{MSet}(A)_n = \sum_{j=1}^{n} \sum_{r=1}^{[n/j]} \text{MSet}(A_{j}, \text{card} = r) \cdot \text{MSet}(A_{>j})_{n-r} - [\text{MSet}(A)_n \neq \emptyset], \]
as we wanted to prove. A similar reasoning applies to powersets.

Since (16) implies the weaker statement \( A\mathcal{A} = cA(z) + o(A(z)) \), this translates immedi-
ately to coefficients, namely, \( A\mathcal{A}_n = cA_n + o(a_n) \).

Finally, we have shown in [15] that for \( C = \text{Cycle}(A) \) the statement \( A\mathcal{C}_n/c_n = \Theta(1) \)
provided that \( A\mathcal{A}_n/a_n \) is also constant. Applying structural induction, we may conclude
that the statement of the theorem holds for whatever unlabelled admissible class. □
5.2. The cost of generating labelled classes

For labelled classes we can set up an $A$-calculus as we have done for unlabelled classes in the previous subsection. We use the same assumptions as before. For the trivial cases we have again $\emptyset = A\emptyset = AZ = 0$, and for disjoint unions we have

$$A(A + B) = \emptyset A + A\emptyset + \emptyset B + B\emptyset - \emptyset (A + B),$$

since the argument used before is still valid for labelled classes.

Now, for labelled products, the cost of generating $(n, j)$-partitions can be evenly distributed among the generated objects, and thus “absorbed” in the remaining terms, much in the same manner as we charge a unitary cost to the non-recursive steps performed each time we call $\text{next}$. Structure-first order is slightly more inefficient than partition-first (because it is more efficient to generate labellings while keeping “constant” the two components of a pair), but we will analyze structure-first generation since it is easier. Since the cost of generating all labelled objects of size $n$ in $C = A_j \otimes B_{n-j}$ is

$$A(A_j \otimes B_{n-j}) = \Theta \left( \binom{n}{j} \right) a_j b_{n-j}$$

$$+ \left( \binom{n}{j} \right) a_j AB_{n-j} + \left( \binom{n}{j} \right) A.A_j \not\emptyset B_{n-j} \neq \emptyset \right)$$

and we can “absorb” the first term corresponding to the generation of labels, the corresponding rule reads

$$A(A \otimes B) = AAB + AAA[BB] + [AA][BB] - [A \otimes B]$$

which again coincides with the rule for unlabelled products.

Finally, for boxed products and using an analogous reasoning we have for $0 < j \leq n$ and $n > 0$

$$A(A_j \boxdot B_{n-j}) = \Theta \left( \binom{n-1}{j-1} \right) a_j b_{n-j}$$

$$+ \left( \binom{n-1}{j-1} \right) a_j AB_{n-j} + \left( \binom{n-1}{j-1} \right) A.A_j \not\emptyset B_{n-j} \neq \emptyset \right).$$

Multiplying by $[n/j]z^n$ and absorbing the cost of generating the labellings, we can derive the rule for $A(A \boxdot \otimes B)$:

$$\vartheta A(A \boxdot \otimes B) = \vartheta AAB + \vartheta AAA[BB] + \vartheta [AA][BB] - \vartheta [A \boxdot \otimes B],$$

where $\vartheta \equiv z[d/dz]$.

Given these rules we can obtain rules for labelled sequences, sets and cycles without too much effort. However, the reader will notice that we can also establish here that $A.A = A - \emptyset A$ for any admissible labelled class. In particular we need to show that this is indeed the case for boxed products, since then the result will hold for sets and cycles.
Let $C = A \Box \star B$. We have then, by structural induction,

$$\vartheta AC = \vartheta AAB + \vartheta A[\vartheta B] + \vartheta [\vartheta A][\vartheta B] - \vartheta [A \Box \star B]$$

$$= \vartheta AB - \vartheta [A \Box \star B] = \vartheta C - \vartheta [C].$$

where the last equality holds since $\vartheta C = \vartheta AB$. Finally, by linearity of $\vartheta$, we conclude $\vartheta C = C - [C]$.

**Theorem 2.** For any labelled admissible class $A$,

$$\mu_{A,n} = \frac{\Lambda A_n}{a_n} = \Theta(1).$$

Generating products and boxed products in PF order needs less work than in SF order; hence, the constant amortized time per generated object also holds in this case—but the constant will be smaller.

### 5.3. Implementation issues

We have implemented all the algorithms described in this paper in MAPLE; we have extensively used the facilities for automatic counting and specification parsing already provided by the combstruct package. Furthermore, we have tried to follow the usual conventions for MAPLE packages in general and for combstruct in particular, so that our programs fit well with their “environment”; also, we wanted that these programs could interact with our generic routines for unranking [13].

We have conducted a few experiments with the implemented programs in order to assess their practical performance. Although not statistically significant, the experiments support the conclusion that the algorithms work in constant amortized time—we have not listed all objects of given large sizes, but say the first million objects of size 100, 200, 300, ... . The constant depends on the class. For instance, it is around four elementary operations per object in the case of binary trees, and close to 20 elementary operations per object in the case of functional graphs. We have also observed that the performance sometimes approaches the characteristic amortized constant performance slowly; a relatively large number of objects needs to be generated before the (constant) preprocessing and initialization costs do not “mask” the constant that corresponds to the generation proper. In other words, if $C_{N,A,n}$ is the cost of generating the first $N$ objects in $A_n$, the experiments show that $C_{N,A,n}/N \rightarrow \mu_{A,n}$ as $N \rightarrow a_n$, but sometimes this convergence is quite slow.

Last but not least, we have incorporated to all our algorithms the possibility of using fingers to improve performance. A finger is a pointer stored in the iterator which points to the last updated internal node in the representation of the current object. When computing the next object we access in one single step that node; most of the times, computing the next object will only need to change the subobject rooted at the node pointed to by the finger, so we can avoid the recursion branching to the right that appears in all the algorithms. If the subobject that we access through the finger is the last in its class then we have to backtrack as much as necessary; hence, we need a pointer to the parent of each node. Also, while
performing a backtrack we will have to update the ascendants of the node which was pointed to by the finger because several of their attributes (e.g., rank, partition, perm, . . . ) will be outdated.

In the case of labelled objects, the iterator stores the “global” labelling of the object and each node stores a pointer to the relevant label: atom nodes point to the corresponding label; internal nodes to the label of their leftmost atom. This allows for an efficient computation of the labelling of the next object, involving no more changes in the global labelling than the number of updated nodes.

6. Conclusions and future work

We have shown in this paper the validity of the generic approach for the problem of exhaustively generating all the objects of a given size in an admissible class. The algorithms work for arbitrarily complex classes built using the admissible operators in constant amortized time per generated object (Theorems 1 and 2), making them attractive alternatives for their inclusion in general combinatorial libraries or for rapid prototyping. Furthermore they nicely combine with the existing framework for counting, random generation and unranking.

A few problems around this work still remain open. We want to conduct larger and more systematic experiments to study the practical performance of the algorithms developed here. Last but not least, it would be interesting to devise a suitable recursive decomposition of unlabelled cycles; lacking such recursive decomposition, we have come up with an efficient algorithm that departs largely in its design from the other algorithms. This difference in its design also appears at the level of its analysis, where an ad-hoc argument was used to establish CAT performance for the generation of unlabelled cycles. Another line of research is to redesign the algorithms so that there is no need to know the cardinality of the involved classes (basically, the important issue is then to find an efficient and alternative way to compute is_last for any admissible combinatorial class).

Acknowledgements

We thank the comments and suggestions made by the referees of the preliminary version of this paper.

References


