# An Efficient Generic Algorithm for the Generation of Unlabelled Cycles[⋆]

Conrado Martínez[1] and Xavier Molinero[1]

Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08034 Barcelona, Spain.{conrado,molinero}@lsi.upc.es

**Abstract.** In this paper, we combine two recent generation algorithms to obtain a new algorithm for the generation of unlabelled cycles. Sawada's algorithm [12] lists all $k$-ary unlabelled cycles with fixed content, that is, the number of occurences of each symbol is fixed and given *a priori*. The other algorithm [8], by the authors, generates all multisets of objects with given total size $n$ from any admissible unlabelled class $\mathcal{A}$. By admissible we mean that the class can be specificied using atomic classes, disjoints unions, products, sequences, (multi)sets, etc. The resulting algorithm, which is the main contribution of this paper, generates all cycles of objects with given total size $n$ from any admissible class $\mathcal{A}$. Given the generic nature of the algorithm, it is suitable for inclusion in combinatorial libraries and for rapid prototyping. The new algorithm incurs constant amortized time per generated cycle, the constant only depending in the class $\mathcal{A}$ to which the objects in the cycle belong.

## 1 Introduction

The generation of unlabelled cycles (also called *necklaces*) probably poses the most difficult problems if we compare them with the generation of other common combinatorial constructs (see for instance [9, 4, 10]). But in the last few years there has been several notable progress related to the generation of necklaces, as witnessed by the work of Wang and Savage [16], Ruskey and Sawada [13, 11], Cattell *et al.* [1] and Sawada [12].

Recall that a necklace or unlabelled cycle is a sequence of symbols such that it is lexicographically smaller than any of its circular permutations. Thus, `abadc` is a cycle but `adcab` is not. Of course, we assume a certain order among the symbols in order to properly define the notion of cycle. When a cycle is aperiodic it is called a *primitive cycle* or *Lyndon* word; thus the cycle `abadc` is a Lyndon word, but the cycle `abab` is not.

In our recent works [6, 8] we have shown how to design general algorithms to iterate through all the objects of a given size in labelled and unlabelled *admissible combinatorial* classes, such as those constructed using disjoint unions, products, sets and multisets, sequences, substitutions, etc. We use the adjective general above in the sense that the algorithms receive as their input both the size $n$ of the objects and a finite

[⋆] This research was supported by the Future and Emergent Technologies programme of the EU under contract IST-1999-14186 (ALCOM-FT) and the Spanish "Ministerio de Ciencia y Tecnología" programme TIC2002-00190 (AEDRI II).

specification of the combinatorial class which the objects belong to. In this paper we present an iteration algorithm for unlabelled cycles thus extending the framework already presented in [8]. Contrary to the other algorithms that we have designed so far, the algorithm for unlabelled cycles is not based upon a suitable recursive decomposition of this combinatorial construction; hence, it cannot be used as the basis for an efficient *unranking* algorithm (that is, given a rank $i$ and a size $n$, generate the $i$-th object of size $n$) for unlabelled cycles, which remains still as an open problem.

The framework presented in [6, 8] follows the approach that was pionereed by Flajolet *et al.* [3] for the random generation of combinatorial objects, and later applied by the authors to the unranking of combinatorial objects [7]. Together with the present paper, these papers show the viability of this elegant approach for an effective and efficient solution of the "big four": counting, random generation, exhaustive/iterative generation and unranking. For instance, a typical implementation of these iteration algorithms could be used to generate and print all cycles of positive integers whose sum is 10, as follows:

```
neck:= [ N,  N = Cycle(Set(Z, card >= 1)) , unlabelled ];
# Z denotes a generic atomic class
it:= init_iter(neck, size = 10);
while not is_last(it) do print(get_obj(it)); it:= next(it); end
```

We have been able to prove that all the iteration algorithms proposed in [6, 8] run in constant amortized time (CAT) per generated object, provided that the class can be finitely specified using $\epsilon$-classes (a single object of size 0), atomic classes (a single object of size 1), disjoint unions, products, sequences, substitutions, sets and multisets. We also provided there a CAT algorithm for labelled cycles, but unlabelled cycles eluded our efforts. The proposed algorithms do not perform better than state-of-the-art algorithms, but do not perform much worse either. And because of their general flavor, they are useful for rapid prototyping and for their inclusion into general combinatorial libraries like `combstruct` for Maple or `MuPAD-combinat` for MuPAD.

This paper is organized as follows. In Section 2 we review a few basic concepts and notations. Sections 3 and 4 present our algorithm, which is based upon our algorithm for unlabelled multisets in [8] and the recent algorithm to generate unlabelled cycles of fixed content of Sawada [12]. We prove that the resulting algorithm has good performance, namely, it is constant amortized time. Finally, in section 5 we discuss our current and future work on this topic and report on our preliminary implementation of the algorithm.

## 2 Preliminaries

In this paper we consider *unlabelled admissible classes of combinatorial structures* and in particular, unlabelled cycles. Most of the material in this section is standard and can be found elsewhere, see for instance [2, 14, 15]. However, to make the paper more self-contained and to fix notation, we will briefly introduce some basic definitions and concepts. We begin with the formal definition of a combinatorial class.

**Definition 1** *A* combinatorial class *is a pair* $(\mathcal{A}, |\cdot|)$ *such that* $\mathcal{A}$ *is a finite or infinite denumerable set and* $|\cdot| : \mathcal{A} \to \mathbb{N}$ *is a* size *function such that, for all* $n \geq 0$*,* $\mathcal{A}_n = \{\alpha \in \mathcal{A} \,|\, |\alpha| = n\}$ *is finite.*

Shall no confusion arise, we will use the same name for the class and for the set of objects belonging to that class. Also, we use subscripts under a class name to denote the subset of objects of that class with a given size. Typically, complex objects in a given class are composed by smaller units, called *atoms* and generically denoted by $Z$. Atoms are objects of size 1 and the size of an object is the number of atoms it contains. For instance, a string is composed by the concatenation of symbols, where each of these is an atom, and the size of the string is its length or the number of symbols it is composed of. Similarly, a tree is built out of nodes —its atoms— and the size of the tree is its number of nodes. Objects of size $0$ are normally denoted by $\epsilon$[1].

Two main types of combinatorial classes can be defined depending on whether the atoms that compose a given object can be distinguished or not. In the former case, we say the class is *labelled* whereas in the later we say the class is *unlabelled*.

As it will become apparent, an efficient solution to the problem of counting, namely, given a specification, a class and a size, compute the number of objects of the given size, is fundamental for our solution of the iteration problem. Hence, we turn our attention to *admissible combinatorial classes*, that is, those classes that can be constructed from *admissible constructors*. An admissible constructor is an operation over classes that yields a new class, and such that the number of objects of a given size in the new class can be computed from the number of objects of that size or smaller sizes in the constituent classes.

In order to formalize the notion of admissibility, we need the fundamental notion of *counting generating functions*.

**Definition 2** *The (counting) generating function of an unlabelled combinatorial class* $\mathcal{A}$ *is the ordinary generating function for the sequence* $\{a_n\}_{n\geq 0}$*. That is,*

$$A(z) = \sum_{n\geq 0} a_n z^n = \sum_{\alpha\in\mathcal{A}} z^{|\alpha|}$$

*where* $a_n = \#\mathcal{A}_n$ *is the number of objects in* $\mathcal{A}$ *of size* $n$*. The* $n$*-th coefficient of* $A(z)$ *is denoted* $[z^n]A(z)$*, i.e.,* $a_n = [z^n]A(z)$*.*

Now we can define *admissible operators*.

**Definition 3** *An operation* $\Psi$ *over combinatorial classes* $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ *is* admissible *if and only if there exists some operator* $\Phi$ *over the corresponding generating functions* $A_1(z), \ldots, A_k(z)$ *such that*

$$\mathcal{C} = \Psi(\mathcal{A}_1, \ldots, \mathcal{A}_k) \implies C(z) = \Phi(A_1(z), \ldots, A_k(z))$$

*where* $C(z)$ *is the generating function of* $\mathcal{C}$*.*

Finally, we can define admissible specifications.

[1] Some authors use $\lambda$ to denote object of size $0$, also called the empty object.

**Definition 4** *An* admissible specification $S$ *is a collection of equations of the form*

$$\mathcal{A}_i = \Psi_i(\mathcal{A}^{(i)}_{j_0}, \ldots, \mathcal{A}^{(i)}_{j_i}) \ ,$$

*where no two equations have the same left-hand side, each $\Psi_i$ is an admissible operation, and each $\mathcal{A}^{(i)}_{j_r}$ is either an $\epsilon$-class, an atomic class, or there is an equation in the collection with that class as its left-hand side. An $\epsilon$-class is a class that contains a single object of size $0$. Each of the classes that appear in the left-hand sides of the equations in $S$ is said to be specified by $S$.*

If a class $\mathcal{A}$ is specified by an admissible specification, then the class itself is called admissible. Admissible classes are also called *decidable* or *well-founded* classes [17]. Table 1 lists some admissible operators and the corresponding ordinary generating functions (OGFs).

| **Class** | **OGF** |
|---|---|
| $\mathsf{Union}(\mathcal{A},\mathcal{B}) = \mathcal{A}+\mathcal{B}$ | $A(z)+B(z)$ |
| $\mathsf{Prod}(\mathcal{A},\mathcal{B}) = \mathcal{A}\times\mathcal{B}$ | $A(z)\cdot B(z)$ |
| $\mathsf{Seq}(\mathcal{A})$ | $\frac{1}{1-A(z)}$ |
| $\mathsf{PowerSet}(\mathcal{A})$ | $\exp\left(\sum_{n>0}(-1)^{n-1}\frac{A(z^n)}{n}\right)$ |
| $\mathsf{Set}(\mathcal{A})$ | $\exp\left(\sum_{n>0}\frac{A(z^n)}{n}\right)$ |
| $\mathsf{Cycle}(\mathcal{A})$ | $\sum_{n>0}\frac{\phi(n)}{n}\log\left(\frac{1}{1-A(z^n)}\right)$ |
| $\mathsf{Subst}(\mathcal{A},\mathcal{B})$ | $A(B(z))$ |

**Table 1.** Unlabelled admissible combinatorial operators

Though the collection of operations given above is small, it can be used to describe many important and useful combinatorial classes (see Table 2). Unfortunately, not all operations are admissible, for instance, intersections and differences.

Such admissible classes must have a finite number of objects for any size, i.e., $a_n < \infty$ for any $n \in \mathbb{N}$. Hence, some restrictions over the classes are needed. For instance, $\mathsf{Seq}(\mathcal{A})$ and $\mathsf{Set}(\mathcal{A})$ require that $a_0 = 0$; and, $\mathsf{Subst}(\mathcal{A},\mathcal{B})$ requires that either $b_0 = 0$ or $\mathcal{A}$ is finite.

From now on, by an *admissible class* we mean that the class can be finitely specified using the $\epsilon$ class (the class with a single object of size 0), atomic classes (classes that contain a single object of size 1), disjoint unions (Union or '+'), products (Prod or '$\times$'), sequences (Seq), multisets (Set), powersets (PowerSet) and cycles (Cycle) of admissible classes. Furthermore, the techniques and results presented can be easily extended to variants of the cycle operator which are admissible operators, in particular, to cycles with a restricted number of components. As we have already mentioned in the

| **Unlabelled class** | **Specification** |
|---|---|
| Integer partitions | $\mathcal{A} = \mathsf{Set}(\mathsf{Seq}(Z, \text{card} \geq 1))$ |
| Binary sequences | $\mathcal{B} = \mathsf{Seq}(Z + Z)$ |
| Rooted unlabelled trees | $\mathcal{D} = Z \times \mathsf{Set}(\mathcal{D})$ |
| Non plane binary trees | $\mathcal{E} = Z + \mathsf{Set}(\mathcal{E}, \text{card} = 2)$ |
| Unlabelled hierarchies | $\mathcal{F} = Z + \mathsf{Set}(\mathcal{F}, \text{card} \geq 2)$ |
| Random mapping patterns | $\mathcal{G} = \mathsf{Set}(\mathsf{Cycle}(\mathcal{D}))$ |
| 2-3 trees | $\mathcal{H} = Z + \mathsf{Subst}((Z \times Z) + (Z \times Z \times Z), \mathcal{H})$ |
| Integer partitions without repetition | $\mathcal{I} = \mathsf{PowerSet}(\mathsf{Seq}(Z, \text{card} \geq 1))$ |

**Table 2.** Examples of admissible unlabelled classes

introduction, in [6, 8] we have developed efficient algorithms to cope with all the combinatorial operators mentioned in the list above, except for unlabelled cycles ($\mathsf{Cycle}$). In this paper, we fill that gap.

## 3 Generating Cycles: The Fundamentals

Let $\mathcal{A}$ be some admissible unlabelled class and assume that we already have an efficient procedure `next` to list all objects of a given size $n$ in $\mathcal{A}$, one at a time. How do we list all the objects in $\mathcal{C} = \mathsf{Cycle}(\mathcal{A})$ of a given size?

Roughly, our algorithm to list $\mathcal{C}_n$ works as follows:

Procedure GENCYC:

1. Generate the next multiset $\gamma$ of $\mathcal{A}$'s of size $n$.
2. Generate all valid cycles that can be constructed with the components (possibly repeated) of $\gamma$.
3. Go to step 1.

In order to generate all the cycles using the components of the *basis multiset* $\gamma$ we need to introduce some order among them. If two components $a_1$ and $a_2$ have the same size then their order is given by respective ranks in $\mathcal{A}$ (i.e., the order in which they are listed by the known `next` procedure). Otherwise, the object of smaller size is considered smaller than the other object. We write $a_1 \preceq a_2$ to denote that $a_1$ is smaller than or equal to $a_2$, in the sense above.

On the other hand, the order of multisets could be similarly defined. Let

$$\gamma = \{a_0 \bullet n_0, a_1 \bullet n_0, \ldots, a_{k-1} \bullet n_{k-1}\},$$
$$\gamma' = \{a'_0 \bullet n'_0, a'_1 \bullet n'_1, \ldots, a'_{j-1} \bullet n'_{j-1}\},$$

where the notation $a \bullet n$ indicates that the object $a$ appears $n > 0$ times in the multiset[2]. Assume, furthermore, that each multiset is presented in sorted order, namely, $a_0 \preceq$

[2] We will usually omit the notation $\bullet n$ if $n = 1$, though.

$a_1 \preceq \cdots \preceq a_{k-1}$, and similarly $\gamma'$. Then the relative order of $\gamma$ and $\gamma'$ follows from the obvious lexicographic order; thus $\gamma \prec \gamma'$ if they have a common "prefix" of length $\ell - 1$ and either $a_\ell \prec a'_\ell$ or $a_\ell = a'_\ell$ and $n_\ell > n'_\ell$.

Notice that the procedure GENCYC will generate all cycles of size $n$ in $\mathsf{Cycle}(\mathcal{A})$ if we find a way to solve its two main steps; however, the cycles would not be generated in lexicographic order. For instance, take the basis multisets $\gamma = \{a \bullet 3, b, c\}$ and $\gamma' = \{a \bullet 2, b \bullet 3\}$, with $\gamma \preceq \gamma'$. However, the cycle `abaac` will be generated before than the cycle `aabbb`, yet the latter is lexicographically smaller than the former. If we defined the order between multisets in such a way that $\gamma' \preceq \gamma$, we would produce `ababb` before `aaabc`, so the problem is inherent to the structure of our algorithm, not to the way we have defined the order of multisets.

Our next step is to define a suitable representation of cycles, so that GENCYC can be efficiently implemented. The representation of cycles is conditioned by our already existing framework for the generation of other combinatorial objects; some modifications need to be introduced into that framework in order to accommodate the generation of cycles.

We use a binary tree and some supplementary information to represent any object of a combinatorial class. We call the resulting data structure an *iterator*. Leaves in the binary tree contain the atoms and $\epsilon$ components of the object; the internal nodes are labelled by operators: '+', '$\times$', etc. Each internal node in contains information about the subobject represented by the corresponding subtree. In particular, each node holds the size of the subobject, the specification of the class that the object belongs to, the rank of the object within its class, and the number of objects of that size in the class. Additional fields are used for some particular nodes, as we describe below. Furthermore, each node has pointers to its children and to its parent, and the iterator has a pointer which always points to the last updated node, so that the generation of the next object can be speeded further.

In order to be able to cope with the nested generation of cycles, a 'cycle' node is similar to a 'multiset' node (we will describe these in detail later) and the object represented by the subtree beneath is what we have called the basis multiset; the 'cycle' node contains also an auxiliary structure that establishes how the components of the basis multiset are arranged in the current cycle. While generating all the cycles with a common basis multiset, the subtree below the 'cycle' node gets untouched, only the auxiliary structure is modified. When all such cycles are generated, the algorithm to generate the next multiset is applied to the 'cycle' node, making the appropriate changes in the subtree and updating the auxiliary structure associated to the 'cycle' node, as necessary.

## 4 Generating Cycles: The Details

The generation of multisets is based in the following useful recursive decomposition:

$$\Theta\mathsf{Set}(\mathcal{A}) = \Delta\Theta\mathcal{A} \times \mathsf{Set}(\mathcal{A}), \qquad (1)$$

where $\Theta\mathcal{B}$ denotes the *marking* or *pointing* of the class $\mathcal{B}$ and $\Delta\mathcal{B}$ denotes the *diagonal* or *stack operator* of the class $\mathcal{B}$. The class $\mathcal{C} = \Theta\mathcal{B}$ is obtained by marking each atom of each of its objects, thus producing $n$ different objects for each object of size $n$ in $\mathcal{B}$. The operator $\Theta$ is admissible since $C(z) = \Theta B(z) = z\frac{d}{dz}B(z)$. The class $\mathcal{C} = \Delta\mathcal{B}$ is obtained by forming tuples of repetitions of the objects in $\mathcal{B}$; thus

$$\mathcal{C} = \mathcal{B} + \{(\beta,\beta) \,|\, \beta \in \mathcal{B}\} + \{(\beta,\beta,\beta) \,|\, \beta \in \mathcal{B}\} + \cdots ,$$

We have $C(z) = \Delta B(z) = B(z)+B(z^2)+B(z^3)+\cdots$. An intuitive way to express the isomorphism in (1) is that a multiset consists of some distinguished element repeated a certain number of times, times a multiset. In our representation, a 'multiset' node has a left son of size $\ell$ whose root is a 'delta' node and a right son of size $n-\ell$ whose root is a 'multiset' node. To compute the next multiset, the algorithm is recursively applied in the right son; but if no more multisets of size $n-\ell$ can be generated, then we apply the `next` algorithm to the left son and initialize the right son with the smallest multiset of size $n-\ell$ that does not contain components smaller or equal to the object in the 'delta' node. If there is no 'delta' object following the one represented by the current left son, then we initialize both the left son and the right son with the appropriate 'delta' and 'multiset' objects of size $\ell' > \ell$ and $n-\ell'$, respectively.

The computation of the next 'delta' object is quite easy. Suppose that the current 'delta' object is $a \bullet r$, of size $\ell = |a| \cdot r$. The 'delta' node has a special field to carry the information about the number of repetitions $r$ of the object $a$, which is represented in the left subtree. The right subtree of the 'delta' node is simply discarded. If $a$ is not the last object in $\mathcal{A}$ of its size, then the next 'delta' object is $a' \bullet r$, where $a'$ is the next object of $a$ in $\mathcal{A}$. Otherwise, find the smallest divisor $r' > r$ of $\ell$ and initialize the 'delta' node with $a_0 \bullet r'$ where $a_0$ is the first object of size $\ell/r'$ in $\mathcal{A}$. With this scheme, multisets are not generated in lexicographic order, but as we have discussed, this is not very relevant as the cycles can't be generated in lexicographic order either.

In order to generate a multiset that does not contain a given component (nor any smaller component), we use the rank and size of the $\mathcal{A}$ object which must be avoided when initializing the multiset. That means that, recursively, its 'delta' node starts with the object following the given one, either of the same size and of given rank plus one, or of the next available size.

Provided that all objects of a given size in $\mathcal{A}$ can be generated in time proportional to the total number of generated objects, then it can be shown that the algorithm above generates all multisets of size $n$ of $\mathcal{A}$'s in time proportional to the total number of generated multisets [8].

We use Sawada's algorithm [12] to generate all the cycles with a given basis multiset. This recent algorithm is able to produce all possible cycles that contain $N_0$ occurrences of the symbol $0$, $N_1$ occurrences of symbol 1, etc., i.e., all $k$-ary cycles of *fixed content*. In our algorithm, the components of the basis multiset are the "symbols" to work with. With some effort, Sawada's algorithm can be transformed into an iterative version which is more convenient to our purposes, although more involved than the original recursive version. The algorithm works by appending symbols to suitable prefixes; in order to avoid linear searches of the available symbols, a global list $L$ of "available" symbols is used.

To start the generation of cycles, we first initialize the basis multiset $\gamma$ of the 'cycle' object, along the lines sketched above. Let $\gamma = \{a_0 \bullet r_0, \ldots, a_{k-1} \bullet r_{k-1}\}$. We also prepare a list $L$ with pointers to the 'delta' objects $a_i \bullet r_i$ and initialize an array $N$ with the repetitions $r_i$ of each 'delta' object. Both the list $L$ and the array $N$ are part of the auxiliary structure attached to the 'cycle' node. Because of the requirements of Sawada's algorithm the list $L$ must be filled in reverse order (the first element in the list points to $a_{k-1}$ and so on) and then $L$ must be rearranged to make sure that the first element in $L$ points to the component with the largest value $r_i$. Of course, the array $N$ must be rearranged accordingly. The auxiliary structure also contains the rank of the current cycle (initialized to 0) and the count of the number of cycles that can be generated with that particular basis multiset [5]:

$$C(N_0, \ldots, N_{k-1}) = \frac{1}{N} \sum_{j \,|\, \mathrm{gcd}(N_0,\ldots,N_{k-1})} \phi(j) \frac{(N/j)!}{(N_0/j)! \cdots (N_{k-1}/j)!},$$

where $\phi(x)$ is Euler's totient function (the number of prime divisors of $x$) and $N = N_0 + N_1 + \cdots + N_{k-1}$ (observe that $N$ is not necessarily the size of the cycle, as the components are not of size 1, in general). Finally, it contains an array $V$ of pointers that gives the arrangement of the components within the current cycle, the length $p$ of the longest Lyndon prefix of the current cycle, as well as other necessary bookkeeping variables. All this additional information, in particular $L$ and $N$, can be easily initialized while initializing the basis multiset, introducing a few minor modifications in the corresponding algorithm.

Recall that the list $L$ is initialized in reverse order and furthermore, Sawada's algorithm requires that symbols are renamed to make sure that the largest symbol in lexicographic order has the largest number of occurrences. Although it would be not too difficult to take this circumstances into account when accessing the binary trees and associated data structures which represent the object and undo both the reversing and renaming, we do not take any further steps on this respect, because we will not be able to generate all cycles of $\mathcal{A}$'s in lexicographic order anyway.

When all the cycles with the same basis multiset are exhausted, the `next` algorithm is applied to obtain a new basis multiset; the list $L$, the array $N$ and the other fields of the auxiliary structure are updated accordingly. In principle, updating the array $N$ and the list $L$ would need some non-constant amount of work. But when Sawada's algorithm ends, both $L$ and $N$ have recovered their initial contents. Again a minor modification of the `next` algorithm will allow us to make no more changes to update $N$ and $L$ than to generate the new basis multiset (in the amortized sense). So to speak, most of the times only one or two components of the multiset will be modified and thus only a constant amount of work will be necessary to update $N$ and $L$.

Sawada's algorithm generates the $C(N_0, \ldots, N_{k-1})$ cycles in constant amortized time per cycle. Denote $C(\gamma)$ the cost of generating all the cycles with basis multiset $\gamma$ and $\mathfrak{C}(\gamma)$ the set of cycles with basis multiset $\gamma$. Thus the cost $C_n$ of our algorithm is given by:

$$\begin{aligned} C_n &= c \cdot \#\mathsf{Set}(\mathcal{A})_n + \sum_{\gamma \in \mathsf{Set}(\mathcal{A})_n} (C(\gamma) + c') \\ &= (c + c') \cdot \#\mathsf{Set}(\mathcal{A})_n + \sum_{\gamma \in \mathsf{Set}(\mathcal{A})_n} \sum_{v \in \mathfrak{C}(\gamma)} c'' \\ &= c''' \cdot \#\mathsf{Set}(\mathcal{A})_n + c'' \cdot \#\mathsf{Cycle}(\mathcal{A})_n, \end{aligned}$$

where $c$, $c'$, $c''$ and $c'''$ are constants. The constant $c$ is implied by our CAT algorithm for multisets; the constant $c''$ is implied by Sawada's algorithm for cycles of fixed content, and $c'$ is the constant amount of work that needs to be done to update the auxiliary structure attached to the corresponding 'cycle' node each time that we move from a basis multiset onto the next one. Since $\mathsf{Set}(\mathcal{A})_n \leq \mathsf{Cycle}(\mathcal{A})_n$ if $n > 0$, it follows that $C_n$ is bounded by a constant times the number of generated objects, and thus it is a CAT algorithm too.

## 5 Final Remarks

We have already conducted a few experiments with a preliminar implementation of the algorithm described in this paper in MAPLE with good results. In particular, we have used the class $\mathcal{N} = \mathsf{Cycle}(\mathsf{Set}(Z, \text{card} \geq 1))$ (cycles of integers), for our experiments. For instance, all cycles of integers of total size 25—there are 1342183 such cycles—are generated in 2269 seconds (0.0016911 seconds/cycle) using a machine equipped with a Pentium processor at 1.7 GHz.

Although the basic ideas behind the algorithms (including the algorithm for multisets and Sawada's algorithm) are rather simple, the implementation details are not. This is because the new algorithm for cycles operates rather differently of the other generation algorithms with which it should be integrated (the algorithm for multisets is prototypical in that respect, with a nice recursive decomposition guiding the algorithms operation). We are looking for a suitable recursive decomposition of cycles that allows us to design an iteration algorithm which fits better the framework developed in [8]. Also, such an algorithm would be more amenable to a precise analysis of its performance. Last but not least, it such a recursive decomposition were obtained, an efficient algorithm for the unranking of cycles would suggest itself.

Other related questions that we are now investigating include variants of the operators (for instance, cycles with restrictions on the number of components) and minor variations of the order in which objects are generated which could improve the overall performance of the process.

## References

1. K. Cattell, F. Ruskey, J. Sawada, and M. Serra. Fast algorithms to generate necklaces, unlabeled necklaces, and irreducible polynomials over GF(2). *J. Algorithms*, 37:267–282, 2000.
2. Ph. Flajolet and R. Sedgewick. The Average Case Analysis of Algorithms: Counting and generating functions. Technical Report 1888, INRIA, April 1993.

3. Ph. Flajolet, P. Zimmerman, and B. Van Cutsem. A calculus for the random generation of combinatorial structures. *Theoret. Comput. Sci.*, 132(1-2):1–35, 1994.
4. H. Fredricksen and I.J. Kessler. An algorithm for generating necklaces of beads in two colors. *Discrete Mathematics*, 61:181–188, 1986.
5. E.N. Gilbert and J. Riordan. Symmetry types of periodic sequences. *Illinois J. Mathematics*, 5:657–665, 1961.
6. C. Martínez and X. Molinero. Generic algorithms for the exhaustive generation of labelled objects. In *Proc. of the 4th Workshop on Random Generation of Combinatorial Structures and Bijective Combinatorics (GASCOM'01)*, pages 53–58, 2001.
7. C. Martínez and X. Molinero. A generic approach for the unranking of labelled combinatorial classes. *Random Structures & Algorithms*, 19(3–4):472–497, 2001.
8. C. Martínez and X. Molinero. Generic algorithms for the generation of combinatorial objects. In *Proc. of the 28th Int. Symposium on Mathematical Foundations of Computer Science (MFCS)*, Lecture Notes in Computer Science, 2003. Accepted for publication.
9. A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms*. Academic Press, 1978.
10. F. Ruskey, C. Savage, and T.M.Y. Wang. Generating necklaces. *J. Algorithms*, 13:414–430, 1992.
11. F. Ruskey and J. Sawada. A fast algorithm to generate unlabeled necklaces. In *Proc. of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 256–262, 2000.
12. J. Sawada. A fast algorithm to generate necklaces with fixed content. *Theoret. Comput. Sci.*, 2003. To appear.
13. J. Sawada and F. Ruskey. An efficient algorithm for generating necklaces with fixed density. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–758, 1999.
14. R. Sedgewick and Ph. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading, MA, 1996.
15. J.S. Vitter and Ph. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9. North-Holland, 1990.
16. T.M.Y. Wang and C. Savage. A Gray code for necklaces of fixed density. *SIAM J. Discrete Math.*, 9(4):654–673, 1996.
17. P. Zimmermann. *Séries génératrices et analyse automatique d'algorithmes*. PhD thesis, École Polytechnique, March 1991.