Generic Algorithms for the Generation of Combinatorial Objects*

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. This paper briefly describes our generic approach to the exhaustive generation of unlabelled and labelled combinatorial classes. Our algorithms receive a size n and a finite description of a combinatorial class $\mathcal{A}$ using combinatorial operators such as union, product, set or sequence, in order to list all objects of size n in $\mathcal{A}$. The algorithms work in constant amortized time per generated object and thus they are suitable for rapid prototyping or for inclusion in general libraries.

1 Introduction

Exhaustively generating all the objects of a given size is an important problem with numerous applications that has attracted the interest of combinatorialists and computer scientists for many years. There is a vast literature on the topic and many ingenious techniques and efficient algorithms have been devised for the generation of objects of relevant combinatorial classes (permutations, trees, sets, necklaces, words, etc.). Indeed, it is common to find introductory material in many textbooks on algorithms (see for instance [8]). Furthermore, several distinct natural (and useful) orderings have been considered for the generation of combinatorial classes, for example, lexicographic and Gray ordering. Many state-of-the-art algorithms for exhaustive generation can be found (and executed) in the Combinatorial Object Server (`www.theory.csc.uvic.ca/~cos`), where the interested reader can also find further references.

The ultimate goal is to achieve algorithms with constant amortized time per generated object, that is, the cost of generating all N objects of size n takes time proportional to N. Many such algorithms are known, but there is still on-going and active research on this topic.

In this work, we combine some well-known principles and a few novel ideas in a generic framework to design algorithms that solve the problem of exhaustive generation, given the size and a finite specification of the combinatorial class whose elements are to be listed. This kind of approach was pioneered by Flajolet *et al.* [2] for the random generation of combinatorial objects and later applied by the authors for the *unranking* problem [6] and for the generation of labelled

* 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).

objects [5]. Somewhat different, but with a similar spirit, is the general approach of Kemp [4] for the generation of words in lexicographic order.

We show that all our algorithms work in constant amortized time and provide a general framework for the analysis of the performance of these algorithms in the form of a calculus or set of rules.

Most existing algorithms exploit particular characteristics of the combinatorial class to be generated, thus achieving improved performance over naïve or brute force methods. The main contribution of this work is to provide a few generic algorithms which solve the problem of iteration over the subset of objects of a given size, given the size and a finite specification of the combinatorial class. These finite specifications are built from basic ϵ- and atomic classes, and combinatorial operators like unions ('+'), Cartesian products ('$\times$'), sequences ('$\mathfrak{S}$'), multisets ('$\mathfrak{M}$'), cycles ('$\mathfrak{C}$'), etc. Our algorithms, deprived of specific knowledge of the problem at hand, are likely to be a bit worse than their specific counterparts, but still have competitive performance, making them good candidates for rapid prototyping and for inclusion into general combinatorial libraries such as the combstruct package [1] for MAPLE[1] and MuPAD-combinat for MuPAD (`mupad-combinat.sourceforge.net`).

Typically, complex objects in a given class are composed by smaller units, called *atoms*. 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 and 1 will be generically denoted by ϵ and Z, respectively[2]. *Unlabelled* objects are those whose atoms are indistinguishable. On the contrary, each of n atoms of a *labelled* object of size n bears a distinct label drawn from the numbers 1 to n.

For the rest of this paper, we will use calligraphic uppercase letters to denote classes ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\dots$). Given a class $\mathcal{A}$, $\mathcal{A}_n$ will denote the subset of objects of size n in $\mathcal{A}$ and a_n the number of such objects. We use the corresponding uppercase roman letter (A, B, C, ...) to denote the counting generating functions (ordinary GFs for unlabelled classes, exponential GF for labelled classes). The n-th coefficient of $A(z)$ is denoted $[z^n]A(z)$; hence, $a_n = [z^n]A(z)$ if $A(z)$ is ordinary and $a_n = n! \cdot [z^n]A(z)$ if $A(z)$ is exponential.

As it will become apparent, our approach to the exhaustive generation problem requires an efficient algorithm for counting, that is, given a specification of a class and a size, compute the number of objects with the given size. Hence, we will only deal with so-called *admissible combinatorial classes* [10, 9]. Those are

---

[1] The current implementation of combstruct offers a routine `allstructs` to generate all objects of a given size and a finite specification of the class; but it does the job by repeatedly generating objects at random until all them have been generated.

[2] Also, we will use these symbols to denote not only objects but the classes that contain just one object of size 0 and of size 1, respectively.

constructed from *admissible operators*, operations over classes that yield new classes, 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. Tables 1 and 2 give a few examples of both labelled and unlabelled admissible classes; as such, our algorithms are able to generate all their objects of a given size and the specification of the class.

Labelled class	**Specification**
Cayley trees	$\mathcal{A} = Z \star \mathfrak{M}(\mathcal{A})$
Binary plane trees	$\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$
Hierarchies	$\mathcal{C} = Z + \mathfrak{M}(\mathcal{C}, \mathrm{card} \geq 2))$
Surjections	$\mathcal{D} = \mathfrak{S}(\mathfrak{M}(Z, \mathrm{card} \geq 1)))$
Functional graphs	$\mathcal{E} = \mathfrak{M}(\mathfrak{C}(\mathcal{A}))$

Table 1. Examples of labelled classes and their specifications

Any combinatorial object belonging to an admissible class can be suitably represented as a string of symbols or as an expression tree whose leaves correspond to the object's atoms and whose internal nodes are labelled by admissible operators. However, such a representation is not the most convenient for the exhaustive generation problem; our algorithms will act upon a new kind of objects, which we call *iterators*. An iterator contains a combinatorial object (represented as a tree-like structure), but also additional information which helps and speeds up the generation process. This additional information is also organized as a tree-like structure—which we call *deep structure*—and reflects that of the corresponding combinatorial object, but each node contains information about the class, the rank, the size, the labelling of the "subobject" rooted at the node in the case of labelled objects, etc. Furthermore, there are pointers between the object's representation and the deep structure to allow fast access and update of the object's representation.

Unlabelled class	**Specification**
Binary sequences	$\mathcal{A} = \mathfrak{S}(Z + Z)$
Necklaces	$\mathcal{B} = \mathfrak{C}(\mathfrak{M}(Z, \mathrm{card} \geq 1))$
Rooted unlabelled trees	$\mathcal{C} = Z \times \mathfrak{M}(\mathcal{C})$
Non plane ternary trees	$\mathcal{D} = Z + \mathfrak{M}(\mathcal{D}, \mathrm{card} = 3)$
Integer partititons without repetition	$\mathcal{E} = \mathfrak{P}(\mathfrak{S}(Z, \mathrm{card} \geq 1))$

Table 2. Examples of unlabelled classes and their specifications

From the user's point of view, we shall offer the following four routines: 1) a procedure to initialize the iterator (`init_iter`), which given a finite description of a combinatorial class $\mathcal{A}$ and a size n, returns the iterator corresponding to the first object in $\mathcal{A}_n$; 2) a function `next`, which given an iterator modifies it so that it corresponds to the object following the previous one; 3) a function `get_obj` to retrieve the combinatorial object from the iterator, in order to print or process it as needed; 4) a boolean function `is_last` to check whether the iterator corresponds to the past-the-end object: a fictitious sentinel which follows the last object in $\mathcal{A}_n$. These will be typically used as follows:

```
it:= init_iter(𝒜, n);
while not is_last(it) do print(get_obj(it)); it:= next(it); end
```

In next section we will describe our algorithms and their performance for the generation of admissible unlabelled combinatorial classes. Section 3 briefly considers the generation of labelled objects, based on our previous work [5], thus integrating the generation of labelled and unlabelled classes into a single and elegant framework. Except for the case of unlabelled cycles (not describe here), most common combinatorial constructions can be dealt within this framework. In section 4 we comment our current work, extensions and future developments.

2 Unlabelled classes

Here, by an *admissible class* we mean that the class can be finitely specified using the ϵ class (the class with a single object of size 0), atomic classes (classes that contain a single object of size 1), and disjoint unions ('+'), products ('$\times$'), sequences ('$\mathfrak{S}$'), multisets ('$\mathfrak{M}$') and powersets ('$\mathfrak{P}$') of admissible classes. We have also developed a generic algorithm for cycles ('$\mathfrak{C}$') of admissible unlabelled classes; however both the algorithm and its analysis use rather different ideas and techniques from the other operators and will be not explained here because of the space limitations.

There exist other interesting admissible operations, but we shall restrict our attention to those mentioned above. Even within this restricted framework, many important combinatorial classes can be specified. Furthermore, the techniques and results presented here can be easily extended to other admissible operators such as substitutions, sequences, multisets and powersets of restricted cardinality, etc.

2.1 The algorithms

The problem of generating the ϵ class and atomic classes is trivial. We shall only consider the function `next`, as the others functions are more or less straightforward. We assume that we have a function $\texttt{count}(\mathcal{A}, n)$ which returns the number of objects of size n in the combinatorial class $\mathcal{A}$ [2].

The function `next` actually uses a recursive routine which receives a pointer p to some node of the deep structure; the initial recursive call is with p pointing to the root of the deep structure. However, we will use the same name for the recursive routine which actually performs the job.

If the current object (of size n) belongs to a class $\mathcal{A}+\mathcal{B}$, we need only to check whether the current object is the last of $\mathcal{A}$ or not. If it is, then the next object will be the first object in $\mathcal{B}$; otherwise, we generate the next object in the appropriate class ($\mathcal{A}$ if the current rank is smaller than or equal to $\texttt{count}(\mathcal{A}, n) - 1$, $\mathcal{B}$ if the current rank is greater than or equal to $\texttt{count}(\mathcal{A}, n) - 1$). This actually means recursively applying the procedure to the unique subtree hanging from p. All the checks above can be easily done as the node pointed to by p in the deep structure contains the specification, the rank of the current object, its size, etc.

On the other hand, if the current subobject of size n corresponds to a product, say $\mathcal{A}\times\mathcal{B}$, we check if the second component (given by *p.second*) is the last object of size $n-k$ of its class $\mathcal{B}$, where $k = $ *p.first.size*. If it is not, we recursively obtain the next object of the second component. If the second component is the last object of size $n-k$ in $\mathcal{B}$, but the first component (pointed to by *p.first*) is not the last object of size k in $\mathcal{A}$ then the algorithm is recursively applied to the first component; we also reset the information in the current node and set the second component of the new object to be the first object in $\mathcal{B}$ of size $n-k$. If both first and second components were the last objects of sizes k and $n-k$ in $\mathcal{A}$ and $\mathcal{B}$, respectively, then a loop looks for the smallest $k' > k$ such that $\mathcal{A}_{k'} \times \mathcal{B}_{n-k'}$ is not empty. After that, the first objects in $\mathcal{A}_{k'}$ and $\mathcal{B}_{n-k'}$ are generated and the appropriate information is updated in the current node p.

Multisets are dealt with in a similar spirit as products. The basis of the algorithm is the isomorphism $\Theta\mathfrak{M}(\mathcal{A}) = \Delta\Theta\mathcal{A} \times \mathfrak{M}(\mathcal{A})$, where $\Delta\mathcal{A}$ denotes the *diagonal* or *stacking* of the class $\mathcal{A}$, that is, $\Delta\mathcal{A} = \{\alpha \,|\, \alpha \in \mathcal{A}\} + \{(\alpha,\alpha) \,|\, \alpha \in \mathcal{A}\} + \{(\alpha,\alpha,\alpha) \,|\, \alpha \in \mathcal{A}\} + \cdots$, and $\Theta\mathcal{A}$ is the *pointing* (*marking*) of the class $\mathcal{A}$, that is, the class that we obtain by making ℓ copies of each object of size k in $\mathcal{A}$, but marking a different atom in each copy. If we mark an atom in a multiset we might think that we have marked the object that contains the atom, say the m-th copy of the object; on the right hand side we produce the marked object, make m copies of the marked object and attach a multiset.

A multiset γ consists of two parts: a first component $\alpha \in \mathcal{A}$ of size k together with its number of occurrences, say ℓ, and a second component β which is a multiset of size $n - \ell k$. This second component contains objects in $\mathcal{A}$ whose size is less than or equal to k; in the latter case, their rank is strictly smaller than the rank of α (implying that they have been used as the first component previously). In order to get the next object, we check whether there exist multisets satisfying the conditions above, that is, whether there is some object following β. If not, we obtain the object following α in $\Delta\mathcal{A}_{\ell k}$. When the first component in the current object is the last object in $\Delta\mathcal{A}_{\ell k}$, we loop until we find a suitable size $j' > j = \ell k$, and obtain the respective first objects. The generation of the next object of α is also easy: we obtain the object following α in $\mathcal{A}_k$ if it exists; if not, we look for the smaller divisor k' of $j = \ell k$ which is larger than k and produce the first object in $\mathcal{A}_{k'}$ and attach the appropiate number of occurrences $\ell' = j/k'$.

Powersets are generated in the same vein. For a fixed first component α of size j, we produce all powersets of size $n - j$ made up of objects of size smaller than or equal to j and whose rank is strictly smaller than the rank of α; if there are no more such powersets, we recursively apply the procedure to α. If α is the last object of size j then we look for the next available size j' for the first component. The isomorphim here is given by $\Theta\mathfrak{P}(\mathcal{A}) = \Delta^{[\mathrm{odd}]}\mathcal{A} \times \mathfrak{P}(\mathcal{A}) - \Delta^{[\mathrm{even}]}\mathcal{A} \times \mathfrak{P}(\mathcal{A})$, where $\Delta^{[\mathrm{odd}]}$ and $\Delta^{[\mathrm{even}]}$ are like the diagonal operator, but for odd and even numbers of copies, respectively. The proof of this isomorphism is a bit more involved, and exploits the principle of inclusion-exclusion to guarantee that no element is repeated.

On a purely formal basis, we can introduce $\hat{\Delta} = \Delta^{[\mathrm{odd}]} - \Delta^{[\mathrm{even}]}$, so that we could say $\Theta\mathfrak{P}(\mathcal{A}) = \hat{\Delta}\Theta\mathcal{A} \times \mathfrak{P}(\mathcal{A})$. The operator $\hat{\Delta}$ allows for more convenient symbolic manipulations when computing the cost of the algorithms, but has no combinatorial meaning, though.

2.2 The performance

Let $\Lambda\mathcal{A}_n = \sum_{\alpha\in\mathcal{A}_n} \mathrm{cn}(\alpha)$, where $\mathrm{cn}(\alpha)$ denotes the cost of applying `next` to object α. Then $\mu_{\mathcal{A},n} = \Lambda\mathcal{A}_n/a_n$ is the amortized cost per generated object. We will not include in the cost the preprocessing time needed to compute the tables of counts nor the cost of constructing the first object (and associated information) of a given size in a given class. These can be precomputed just once (or computed the first time they are needed and stored into tables for later reuse) and its contribution to the overall performance will be neglected. Also, we will not include the time to parse the specification and transform it to standard form either as this cost does not depend on n.

Lemma 1. *Given an admissible unlabelled class $\mathcal{A}$, let $\Lambda\mathcal{A}(z)$ denote the ordinary generating function of the cumulated costs $\{\Lambda\mathcal{A}_n\}_{n\geq 0}$. Then*

1. $\Lambda\emptyset = \Lambda\epsilon = \Lambda Z = 0$,
2. $\Lambda(\mathcal{A}+\mathcal{B}) = \Lambda\mathcal{A} + \Lambda\mathcal{B} + [\![\mathcal{A}]\!] + [\![\mathcal{B}]\!] - [\![\mathcal{A}+\mathcal{B}]\!]$,
3. $\Lambda(\mathcal{A}\times\mathcal{B}) = \Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] + A\cdot\Lambda\mathcal{B} + [\![\mathcal{A}]\!]\cdot[\![\mathcal{B}]\!] - [\![\mathcal{A}\times\mathcal{B}]\!]$,
4. $\Lambda\Theta\mathcal{A} = \Theta\Lambda\mathcal{A} + \Theta[\![\mathcal{A}]\!] - [\![\Theta\mathcal{A}]\!]$,
5. $\Lambda\Delta\mathcal{A} = \Delta\Lambda\mathcal{A} + \Delta[\![\mathcal{A}]\!] - [\![\Delta\mathcal{A}]\!]$,
6. $\Lambda\Delta^{[t]}\mathcal{A} = \Delta^{[t]}\Lambda\mathcal{A} + \Delta^{[t]}[\![\mathcal{A}]\!] - [\![\Delta^{[t]}\mathcal{A}]\!], \quad t \in \{\mathit{odd},\ \mathit{even}\}$

where $\Theta A(z) \equiv z\frac{dA}{dz}$, $\Theta\mathcal{A}$ denotes the pointing or marking of the class $\mathcal{A}$, $\Delta A(z) \equiv \sum_{k>0} A(z^k)$, $\Delta\mathcal{A}$ denotes the diagonal of the class $\mathcal{A}$, $\Delta^{[odd]}A(z) = \sum_{k>0} A(z^{2k-1})$, $\Delta^{[even]}A(z) = \sum_{k>0} A(z^{2k})$, $\Delta^{[odd]}\mathcal{A}$ and $\Delta^{[even]}$ denote the odd and even diagonals of the class $\mathcal{A}$, respectively, and $[\![\mathcal{A}]\!] = \sum_{n\geq 0}[\![a_n \neq 0]\!]z^n$, with $[\![P]\!] = 1$ if the predicate P is true and 0 otherwise.

Proof. For classes that contain just one item, the cumulated cost is 0, as we do not count the cost of generating the first object. The rule for unions is straightforward, but we must take care to charge the cost corresponding to computing the next of the last element in $\mathcal{A}$; this is accounted for by the terms $[\![\mathcal{A}]\!]+[\![\mathcal{B}]\!]-[\![\mathcal{A}+\mathcal{B}]\!]$. For products, we generate pairs in $\mathcal{A}_k\times\mathcal{B}_{n-k}$ for $k = 0,\ldots,n$. For a fixed object α of size k we generate all objects in $\mathcal{B}_{n-k}$ to form all pairs whose first component is α; since there are a_k objects of size k and we have to consider all possible values of k, this contribution is given by $A \cdot \Lambda\mathcal{B}$. The other main contribution to the cost comes from the generation of all objects in $\mathcal{A}$ with sizes k from 0 to n (and such that there are objects of size $\mathcal{B}_{n-k}$ to form at least a pair). This is given by the term $\Lambda\mathcal{A} \cdot [\![\mathcal{B}]\!]$. The remaining terms account for the application of `next` to the last pair in $\mathcal{A}_k \times \mathcal{B}_{n-k}$ whenever there exist such a pair, but not for the first pair in $\mathcal{A} \times \mathcal{B}$.

The algorithm for the marking is also straightforward: list all elements in $\mathcal{A}$ of size n with the first atom marked, then list all them again but with the second atom marked and so on. The terms $\Theta[\![\mathcal{A}]\!] - [\![\Theta\mathcal{A}]\!]$ account for the cost of passing from the first listing to the second, from the second to the third, etc.

To generate all objects of size n in the diagonal of the class $\mathcal{A}$, recall that we loop through all divisors d of n such that there are objects of size d. For each such d, we list all objects of size d and for each one we attach the number of copies (n/d) that make up the corresponding object in $\Delta\mathcal{A}$. Thus $\Lambda\Delta\mathcal{A}_n = \sum_{d \text{ divides } n}(\Lambda\mathcal{A}_d + [\![a_d \neq 0]\!]) - [\![\Delta\mathcal{A}_n \neq \emptyset]\!]$. The rule for the odd and even diagonals of $\mathcal{A}$ are similarly obtained.

From Lemma 1 we can easily obtain rules for sequences, sets and multisets.

Corollary 1. *Let $\mathcal{A}$ be an admissible class such that $\epsilon \notin \mathcal{A}$ and let A be its counting generating function. Then*

1. *Let $\mathfrak{S}(A) = 1/(1-A)$. Then*

$$\Lambda\mathfrak{S}(\mathcal{A}) = \mathfrak{S}(A) \cdot (1 + (\Lambda\mathcal{A} + [\![\mathcal{A}]\!] - 1)[\![\mathfrak{S}(\mathcal{A})]\!])\,.$$

2. *Let $\mathfrak{M}(A) = \exp(\sum_{k>0} A(z^k)/k)$. Then*

$$\Lambda\mathfrak{M}(\mathcal{A}) = \mathfrak{M}(A) \cdot \int_0^z \frac{dz}{z \cdot \mathfrak{M}(A)}\big(\Delta\Theta(\Lambda\mathcal{A} + [\![\mathcal{A}]\!]) \cdot [\![\mathfrak{M}(\mathcal{A})]\!] - \Theta[\![\mathfrak{M}(\mathcal{A})]\!]\big).$$

3. *Let $\mathfrak{P}(A) = \exp(\sum_{k>0}(-1)^{k-1}A(z^k)/k)$. Then*

$$\begin{aligned}\Lambda\mathfrak{P}(\mathcal{A}) = \mathfrak{P}(A) \cdot \int_0^z \frac{dz}{z \cdot \mathfrak{P}(A)} \Big(&\hat{\Delta}\Theta(\Lambda\mathcal{A} + [\![\mathcal{A}]\!]) \cdot [\![\mathfrak{P}(\mathcal{A})]\!] - \Theta[\![\mathfrak{P}(\mathcal{A})]\!] \\ &+ [\![\Theta\mathfrak{P}(\mathcal{A})]\!] - [\![\Delta^{[odd]}\Theta\mathcal{A} \times \mathfrak{P}(\mathcal{A})]\!] + [\![\Delta^{[even]}\Theta\mathcal{A} \times \mathfrak{P}(\mathcal{A})]\!]\Big)\,.\end{aligned}$$

Proof. It suffices to use the isomorphisms $\mathfrak{S}(\mathcal{A}) = \epsilon + \mathcal{A} \times \mathfrak{S}(\mathcal{A})$, $\Theta\mathfrak{M}(\mathcal{A}) = \Delta\Theta\mathcal{A} \times \mathfrak{M}(\mathcal{A})$ and $\Theta\mathfrak{P}(\mathcal{A}) = \Delta^{[\mathrm{odd}]}\Theta\mathcal{A} \times \mathfrak{P}(\mathcal{A}) - \Delta^{[\mathrm{even}]}\Theta\mathcal{A} \times \mathfrak{P}(\mathcal{A})$, and apply rules 1-6 in the statement of Lemma 1. In the case of multisets and powersets, the rules can be obtained applying Λ to both sides of the isomorphisms given above, inverting Θ and Λ with rule 4, and solving the resulting linear differential equations. In the case of powersets, the sought cost arises from the difference of costs; we have thus $\Lambda\Theta\mathfrak{P}(A) = \Lambda(\Theta\Delta^{[\mathrm{odd}]}\mathcal{A} \times \mathfrak{P}(\mathcal{A})) - \Lambda(\Theta\Delta^{[\mathrm{even}]}\mathcal{A} \times \mathfrak{P}(\mathcal{A}))$.

We have the following theorem that can be easily established from either the rules that we have just derived or directly reasoning about the algorithms.

Theorem 1. *For any unlabelled admissible class $\mathcal{A}$ which can be finitely specified using ϵ, Z, $+$, $\times$, $\mathfrak{S}$, $\mathfrak{M}$ and $\mathfrak{P}$, we have*

$$\mu_{\mathcal{A},n} = \Lambda\mathcal{A}_n / a_n = \Theta(1).$$

Proof (Sketch of proof). The proof is by structural induction on the specification of the class and on the size of the objects. We consider thus what happens for unions, products, sequences, etc. and assume that the statement is true for smaller sizes. Since we charge just one "time" unit for the update of one node in the deep structure and assume that the initialization and calls to the `count` routine are free, we actually have $\mu_{\mathcal{A},n} \to 1$ as $n \to \infty$. In practice, $\mu_{\mathcal{A},n}$ is different for different classes if we take into account that the (constant) overhead associated with each operator varies.

We conclude with a few simple examples of application of these rules.

1. K-shuffles. A K-shuffle is a sequence of a's and b's that contains exactly K b's. Let $\mathcal{L}_K = \mathfrak{S}(a) \times (b \times \mathcal{L}_{K-1})$ for $K > 0$ and $\mathcal{L}_0 = \mathfrak{S}(a)$. It is not difficult to show that $\Lambda\mathcal{L}_K \sim \frac{z^K}{(1-z)^{K+1}}$ near the singularity $z = 1$; hence the amortized cost $\mu_{\mathcal{L}_K,n} \to 1$ since there are exactly $\binom{n}{K}$ K-shuffles of size n. We get the same asymptotic performance if we use alternative specifications, e.g., $\mathcal{L}_K = \mathcal{L}_{K-1} \times (b \times \mathfrak{S}(a))$.
2. Motzkin trees. For $\mathcal{M} = \epsilon + Z \times \mathcal{M} + Z \times \mathcal{M} \times \mathcal{M}$, one readily gets $\Lambda\mathcal{M} \sim -1/2(3 - 2\sqrt{2})\sqrt{1 - 6z + z^2} + O((1 - 6z + z^2)^{3/2})$ near the dominant singularity at $z = 3 - 2\sqrt{2}$; hence $\Lambda\mathcal{M} \sim M$ and $\mu_{\mathcal{M},n} = 1 + o(1)$.

3 Labelled classes

By *admissible labelled classes* we mean those that can be finitely specified using the ϵ-class, atomic labelled classes, unions ($+$), labelled products ($\star$), sequences (Seq), sets (Set) and cycles (Cycle) of admissible labelled classes. As in the previous section, there exist other admissible operators over labelled classes, but we shall restrict our attention to those mentioned above. Again, many important combinatorial classes can be specified within this framework and the ideas that we present here carry on to other admissible operators such as substitutions, sequences, sets and cycles of restricted cardinality, etc. For example, the class $\mathcal{C}$ of Cayley trees is admissible, as it can be specified by $\mathcal{C} = Z \star \mathsf{Set}(\mathcal{C})$, where Z denotes an atomic class. The class $\mathcal{F}$ of functional graphs is also admissible, since a functional graph is a set of cycles of Cayley trees; therefore, $\mathcal{F} = \mathsf{Set}(\mathsf{Cycle}(\mathcal{C}))$.

The exhaustive generation of labelled combinatorial objects uses similar ideas to those sketched in Section 2; in fact, we face easier problems since sets and cycles can be specified by means of the so-called *boxed product*, denoted by $^{\square}\!\star$ [3]. We recall that in a boxed product, we obtain a collection of labelled objects from a pair of given objects, much in the same manner as for the usual labelled product, but the smallest label must always correspond to an atom belonging to the first object in the pair. Boxed products are related to the pointing (see subsection 2.1) of a class $\mathcal{A}$ by $\Theta\mathcal{A} = \Theta\mathcal{B}\star\mathcal{C} \iff \mathcal{A} = \mathcal{B}^{\square}\star\mathcal{C}$. The isomorphisms for sequences, sets and cycles in terms of the other constructors (union, product and boxed product) that we use in our algorithms are the following: 1) $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A}\star\mathsf{Seq}(\mathcal{A})$, 2) $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A}^{\square}\star\mathsf{Set}(\mathcal{A})$, and 3) $\mathsf{Cycle}(\mathcal{A}) = \mathcal{A}^{\square}\star\mathsf{Seq}(\mathcal{A})$. Thus every admissible specification (a finite set of equations specifying admissible classes, like in the example of functional graphs) can be transformed into an equivalent specification that involves only unions, products and boxed products.

The algorithms for unions and products are very similar to those for unlabelled classes, and the algorithm for boxed products works much like the algorithm for products. In the case of labelled and boxed products, we change the partition or relabelling of the current object if possible; otherwise we recursively apply the `next` routine to the second component or the first component of the object. In order to "traverse" the $\binom{n}{k}$ possible partitions of the labels[3] of a pair $\langle\alpha, \beta\rangle$, where n is the size of the objects to be generated and k is the size of the first component of the current object, we use Nijenhuis and Wilf's routine for the next k-subset [7] (alternatively, we can use the algorithm by Kemp [4]).

Also, like for unlabelled classes, we can set up a calculus for the complexity of our algorithms with rules such as

$$\Theta\Lambda(\mathcal{A}^{\square}\star\mathcal{B}) = \Theta\Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] + \Theta A\cdot\Lambda\mathcal{B} + \Theta[\![\mathcal{A}]\!]\cdot[\![\mathcal{B}]\!] - \Theta[\![\mathcal{A}^{\square}\star\mathcal{B}]\!],$$

for boxed products. Here, $[\![\mathcal{A}]\!] = \sum_{n\geq 0}[\![a_n \neq 0]\!]\frac{z^n}{n!}$ and $\Lambda\mathcal{A}$ is the exponential generating function of the total costs to generate all elements of each size. The rules for the other combinatorial constructions are similar in spirit and can be easily derived from the rules for unions, products and boxed products.

We make here the same assumptions as in the analysis of the performance of the algorithms for unlabelled classes; moreover, we take into account the fact that the algorithm for the generation of k-subsets works in constant amortized time [7]. Then it is not difficult to show that this cost can be easily "absorbed" by terms like $A\cdot\Lambda\mathcal{B}$ in the rule for products and there is no need to include a term of the type $c\cdot A\cdot B$.

Using the same techniques as in the proof of Theorem 1 is not hard to establish an analogous result for labelled generation. The detailed account of the complexity calculus for the generation of labelled objects and of the proof of the following theorem will be given in the full version of this extended abstract.

Theorem 2. *For any admissible labelled class $\mathcal{A}$ which can be finitely specified using ϵ, Z, $+$, $\star$, Seq, Set and Cycle, we have*

$$\mu_{\mathcal{A},n} = \Lambda\mathcal{A}_n/a_n = \Theta(1).$$

4 Current and Future Work

As we have mentioned earlier, we already have a constant amortized time algorithm to generate unlabelled cycles of $\mathcal{A}$'s. However, this algorithm for the

[3] There are only $\binom{n-1}{k-1}$ partitions of the labels in the case of boxed products.

generation of unlabelled cycles is based upon techniques quite different from the one here and it doesn't fit nicely in the framework here sketched (in sharp contrast with labelled cycles).

We have implemented all the algorithms described here for the MAPLE system, on top of the basic routines provided by the combstruct package. Also, there are plans for a port of these routines to the MuPAD-combinat package in the near future. Furthermore, we also have routines for the generation of labelled substitutions and for labelled sequences, sets and cycles when their cardinalities are restricted. We have conducted extensive experiments to asses the practical performance of our algorithms. These experiments show that the practical performance is in good agreement to the theoretical predictions (namely, the cost grows linearly with the total number N of generated objects, if N is sufficiently large; the slope of the plot is independent of the size of the objects being generated).

Our current work is now centered in the extension of the techniques presented here to other admissible operators. We also are trying to design an algorithm for unlabelled cycles that fits within the framework here sketched. If we obtained such an algorithm, it would immediately suggest an efficient answer for the unranking of unlabelled cycles, a question that still remains open, to the best of the authors' knowledge.

We are also working on alternative isomorphisms and orderings which could improve the efficiency of the generation algorithms (similar ideas yield significant improvements for the random generation and unranking of objects, see [2, 6]).

References

1. Ph. Flajolet and B. Salvy. Computer algebra libraries for combinatorial structures. *J. Symbolic Computation*, 20:653–671, 1995.
2. 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.
3. D.H. Greene. *Labelled Formal Languages and Their Uses.* PhD thesis, Computer Science Dept., Stanford University, 1983.
4. R. Kemp. Generating words lexicographically: An average-case analysis. *Acta Informatica*, 35(1):17–89, 1998.
5. C. Martínez and X. Molinero. Generic algorithms for the exhaustive generation of labelled objects. In *Proc. Workshop on Random Generation of Combinatorial Structures and Bijective Combinatorics (GASCOM)*, pages 53–58, 2001.
6. C. Martínez and X. Molinero. A generic approach for the unranking of labelled combinatorial classes. *Random Structures & Algorithms*, 19(3–4):472–497, 2001.
7. A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms.* Academic Press, 1978.
8. E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, NJ, 1977.
9. R. Sedgewick and Ph. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, Reading, MA, 1996.
10. 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.