Solving Necklace Constraint Problems ⋆

Pierre Flener and Justin Pearson

Department of Information Technology
Uppsala University, Box 337, SE – 751 05 Uppsala, Sweden
Email: Firstname.Lastname@it.uu.se

Abstract. Some constraint problems have a combinatorial structure where the constraints allow the sequence of variables to be rotated (*necklaces*), if not also the domain values to be permuted (*unlabelled necklaces*), without getting an essentially different solution. We bring together the fields of combinatorial enumeration, where efficient algorithms have been designed for (special cases of) some of these combinatorial objects, and constraint programming, where the requisite symmetry breaking has at best been done statically so far. We design the first search procedure and identify the first symmetry-breaking constraints for the general case of unlabelled necklaces. Further, we compare dynamic and static symmetry breaking on real-life scheduling problems featuring (unlabelled) necklaces.
Keywords: constraint problem, combinatorial structure, necklace, static and dynamic symmetry breaking, rotation schedule

1 Introduction

In combinatorics, a *necklace* of n beads over k colours is the lexicographically smallest element in an equivalence class of the set of k-ary n-tuples under rotations; the underlying symmetry group is the cyclic group C_n acting on the indices. For example, the binary triple 001 is the representative necklace of $\{001, 010, 100\}$. Combinatorial objects are enumerated under some chosen total order. For example, under the lexicographic order, the binary 3-bead necklaces are 000, 001, 011, and 111. If the values (colours) of a tuple are interchangeable, then we speak of *unlabelled tuples* (symmetric group S_k acting on the values) and *unlabelled necklaces* (product group $C_n \times S_k$). For example, under the lexicographic order, the unlabelled binary 3-tuples are 000, 001, 010, and 011, while the unlabelled binary 3-bead necklaces are 000 (representing the necklaces 000 and 111) and 001 (representing the necklaces 001 and 011). Note that the set of unlabelled necklaces is a subset of the intersection of the sets of necklaces and unlabelled tuples. For example, 011 is both a necklace and an unlabelled tuple, but not an unlabelled necklace. The generating functions for counting (unlabelled) necklaces are given in [13], and the sequences of their counts (for $k \leq 6$) can be found in [29].

A *constraint satisfaction problem* (*CSP*) is a triplet $\langle X, D, C \rangle$, where X is a sequence of n variables, D is a set of k possible values for these variables and is called their *domain*, and C is the set of constraints specifying which assignments of values to the variables are solutions. If the constraint set C allows the variable sequence X to be

⋆ This paper revises (in Section 4.3) and extends an earlier version published as [9].

rotated, then a necklace is a combinatorial sub-structure of the CSP and we say that the CSP has *rotation variable symmetry*. If the constraint set C has a domain D containing interchangeable elements, then we say that the CSP has *full value symmetry*. Exploiting such symmetry is important in order to solve a CSP efficiently. For example, compare the ternary object counts in Table 1 (a few pages down) with the values of 3^n.

CSPs with an (unlabelled) necklace as a combinatorial sub-structure are not unusual. For example, Gusfield [16, page 12] states that "circular DNA is common and important. Bacterial and mitochondrial DNA is typically circular, both in its genomic DNA and in additional small double-stranded circular DNA molecules called *plasmids*, and even some true eukaryotes (higher organisms whose cells contain a nucleus) such as yeast contain plasmid DNA in addition to their nuclear DNA. Consequently, tools for handling circular strings may someday be of use in those organisms. Viral DNA is not always circular, but even when it is linear some virus genomes exhibit circular properties". One such problem is studied in [5]. Necklaces occur in coding theory [14], genetics [14], and music [13], while unlabelled necklaces occur in switching theory [13]. We study a real-life problem with (unlabelled) necklaces in scheduling, different from the one in [15].

Note that, throughout this paper, we focus on unconditional (or global) symmetries, that is we do not handle any symmetries that appear during search.

In this paper, we propose to bring together combinatorial enumeration and constraint programming (CP). Very efficient combinatorial enumeration algorithms exist for some of the mentioned combinatorial objects, but not for unlabelled necklaces (except over two colours [4]). These algorithms can be used as CP search procedures for CSPs having those objects as combinatorial sub-structures, thereby breaking a lot of symmetry dynamically. This has also been advocated in [30, 23, 11], say, where CP search procedures are proposed for symmetry groups acting on the values; however, except for [28, 10, 11] not much dynamic symmetry breaking seems to have been done for groups acting on the variables, and hence not for product groups acting on the values and variables. Conversely, CP principles can be used for devising enumeration algorithms for the combinatorial objects where efficient algorithms have remained elusive to date. The contributions of this paper can be summarised as follows:

- Design of an enumeration algorithm, and hence a CP search procedure, for (partially) unlabelled k-ary necklaces (Sections 2 and 4).
- Identification of symmetry-breaking constraints for (partially) unlabelled k-ary necklaces, including ways of generating filtering algorithms for the identified new global constraints (Sections 3 and 4).
- Experiments on real-world problems validating the usefulness of the proposed dynamic and static symmetric-breaking methods for (partially unlabelled) k-ary necklaces (Section 4).

Finally, in Section 5, we conclude and discuss future research.

2 Dynamic Symmetry Breaking

Consider a CSP $\langle X, D, C\rangle$ where X is a sequence of $n \geq 2$ variables and D is a set of $k \geq 1$ domain values. For simplicity of notation, we assume that $D = \{0, \ldots, k-1\}$;

```
 1: procedure uTuple(j, u : integer)
 2: var i : integer
 3: if j > n then
 4:    return true
 5: else
 6:    try all i = 0 to min(u + 1, k − 1) do
 7:       X[j] ← i;
 8:       uTuple(j + 1, max(i, u))
 9:    end
10: end if
```

Algorithm 1: Search procedure for unlabelled tuples [7]

this also has the advantage that the order is obvious whenever we require D to be totally ordered.

2.1 Unlabelled Tuples

If the domain values of D are interchangeable, then we impose a total order on D, and the enumeration algorithm of [7], say, can be used to generate all unlabelled tuples (modulo the full value symmetry). We present it as Algorithm 1 in the style of a search procedure in constraint programming (CP), so that it can interact with any problem constraints. The initial call is $uTuple(1, -1)$. At any time, j is the index of the next variable to be assigned (and $j = n + 1$ when none remains) while u is the largest value used so far (and $u = -1$ when none was used yet). The idea is to try for each variable all the values used so far plus one unused value, since all unused values are still interchangeable at that point. Upon backtracking, the **try all** construct non-deterministically tries all the alternatives, in the given value order (line 6). Each alternative contains the assignment of the chosen value i to the chosen variable $X[j]$ (line 7) and a recursive call for the next variable (line 8). Note that we have fixed the variable order to be from left to right across X, and the tuples are thus generated in lexicographic order; this is an unnecessary restriction, but the reason for this choice will become clear in a few lines. This algorithm takes constant amortised time and space, and the number of objects generated is actually equal to the number of unlabelled tuples.

2.2 Necklaces

If the variable sequence X is circular, then the enumeration algorithm of [4], say, can be used to generate all necklaces (modulo the rotation variable symmetry). We present it as a CP search procedure in Algorithm 2. The initial call is $X[0] \leftarrow 0; necklace(1, 1)$, where $X[0]$ is a dummy element. At any time, j is the index of the next variable to be assigned (and $j = n + 1$ when none remains) while p is the *period*, explained next. The idea is either to try and keep replicating the values at the previous p positions, or to try all larger values with a new period of j. At any time, the prefix $X[1, \ldots, j]$ is a *pre-necklace*, that is a prefix of some necklace, which may however be longer than n. The variable order is necessarily from left to right across X, due to the role of p, and

```
 1: procedure necklace(j, p : integer)
 2: var i : integer
 3: if j > n then
 4:    return n mod p = 0
 5: else
 6:    try all i = X[j − p] to k − 1 do
 7:       X[j] ← i;
 8:       necklace(j + 1, if i = X[j − p] then p else j)
 9:    end
10: end if
```

Algorithm 2: Search procedure for necklaces [4]

the necklaces are thus generated in lexicographic order. This algorithm takes constant amortised time and space, and the number of objects generated is proportional by a constant factor (tending down to $(k/(k-1))^2$ as $n \to \infty$) to the number of necklaces: note that only n-tuples where the period p divides n actually are necklaces (line 4). In other words, not all symmetry is broken at every node of the search tree, and some backtracking is forced (by a constant-time test on p) only at leaf level; at present, loopless or memoryless necklace enumeration remains elusive.

2.3 Unlabelled Necklaces

If the variable sequence X is circular *and* the domain values of D are interchangeable, then a constant-amortised-time enumeration algorithm [4] only exists for generating all *binary* ($k = 2$) unlabelled necklaces (modulo the symmetries). We do not present it here, but instead construct a novel enumeration algorithm for *any* amount of colours. Noting that unlabelled necklaces are a subset of the necklaces (Algorithm 2) that are unlabelled tuples (Algorithm 1), and observing that the control flows of those two algorithms match line by line, the skeleton of an enumeration algorithm for unlabelled necklaces can be obtained simply by "intersecting" those two algorithms, which yields all but lines 7 and 10 of the CP search procedure $uNecklace$ in Algorithm 3. The initial call is $X[0] \leftarrow 0$; $uNecklace(1, 1, -1)$, where $X[0]$ is a dummy element.

We now gradually refine the $probe(j, i, p)$ function (called in line 7), guarding the non-deterministic assignment of value i to the current variable $X[j]$ followed by the continued enumeration.

Leaf Probing. If $probe$ always returns $\mathbf{true}$, then $uNecklace$ will enumerate a superset of the unlabelled necklaces, as their symmetry group is the *product* rather than just the union of the symmetry groups for necklaces and unlabelled tuples. For example, the binary necklace 011 will erroneously be returned, even though it can be transformed into the unlabelled necklace 001 (by first rotating the second position of the circular sequence 011 into first position, giving 110, and then minimally renaming its colours, giving $\underline{110} = 001$); however, the necklace 111 will correctly not be returned, since it is not an unlabelled tuple.

```
 1: procedure uNecklace(j, p, u : integer)
 2: var i : integer
 3: if j > n then
 4:    return n mod p = 0
 5: else
 6:    try all i = X[j − p] to min(u + 1, k − 1) do
 7:       if probe(j, i, p) then
 8:          X[j] ← i;
 9:          uNecklace(j + 1, if i = X[j − p] then p else j, max(i, u))
10:       end if
11:    end
12: end if
13: function probe(j, i, p : integer) : boolean
14: X[j] ← i;
15: if j = n ∧ n mod (if i = X[j − p] then p else j) = 0 then
16:    return ⋀_2^{q=n} X[q, ..., n, 1, ..., q − 1] ≥lex X[1, ..., n]
17: else if j < n then
18:    return ⋀_{q=2}^{j−1} X[j − q + 1, ..., j] ≥lex X[1, ..., q]
19: else
20:    return false
21: end if
```

Algorithm 3: Search procedure for unlabelled necklaces

Consider Table 1, giving the numbers of various combinatorial objects of length n over 3 colours: column 8 counts the unlabelled tuples (sequence A124302 in [29]); column 7 counts the necklaces (fewer than the unlabelled tuples for $n \geq 7$; sequence A1867); column 6 counts the objects when *probe* always returns **true**; column 5 counts the necklaces that are unlabelled tuples, that is the number of objects when *probe* always returns **true** and the period condition is met; and column 2 counts the unlabelled necklaces (sequence A2076), that is the number of objects when probing is actually done. The difference between columns 5 and 7 (or 8) is the gain obtained so far for free by Algorithm 3 over Algorithm 2 (or Algorithm 1), and the difference between columns 5 and 6 is the leaf pruning obtained (in constant time!) by the period condition, but the difference between columns 5 and 2 is the amount of leaf pruning that leaf probing has to do.

The least thing $probe(j, i, p)$ should thus do is to make sure only unlabelled necklaces are enumerated. This is at the latest done when trying to assign the last variable (when $j = n$) of the CSP: at that moment, the entire circular sequence X is known, so *probe* must return **true** if X cannot be transformed (by position rotation and colour renaming) into an object that has already been tried in the enumeration. Since objects are enumerated in lexicographic order (as an inherited feature of the two underlying algorithms), this can be done by checking whether the minimal renaming of every (non-identity) rotation of X is lexicographically larger than or equal to X. Computing the minimal renaming $\underline{Y}$ of an n-tuple Y takes $\Theta(n)$ time, and can be merged into the $O(n)$-time lexicographic comparison; at most $n - 1$ such renamings and comparisons are done, hence this probing takes $O(n^2)$ time at worst. Note that a successful probe

	sequence	probing				sequence	sequence
	A2076:	internal + leaf		leaf only		A1867:	A124302:
n	u-necklaces	$n \bmod p = 0$	leaves	$n \bmod p = 0$	leaves	necklaces	u-tuples
1	1	1	1	1	1	3	1
2	2	2	2	2	2	6	2
3	3	4	5	4	5	11	5
4	6	8	10	10	13	24	14
5	9	15	22	24	36	51	41
6	26	34	48	66	97	130	122
7	53	80	121	172	268	315	365
8	146	196	293	474	732	834	1094
9	369	490	744	1289	2017	2195	3281
10	1002	1267	1920	3560	5552	5934	9842
11	2685	3357	5104	9820	15371	16107	29525
12	7434	8996	13635	27327	42624	44368	88574
13	20441	24403	37030	76108	118731	122643	265721
14	57046	66886	101354	213106	331664	341802	797162
15	159451	184770	279895	598246	929883	956635	2391485

Table 1. Numbers of objects of length n over 3 colours

incurs the highest cost. The algorithmic details are straightforward, so we just write a specification into line 16. Lazy evaluation of the conjunction should be made, returning **false** as soon as one conjunct is false. Also, experiments have revealed that failure is detected earlier on average if the starting positions of the rotations recede from right to left across X.

An improvement of this leaf probing comes from observing what happens when the lowest value, namely $X[j-p]$, is tried for $X[j]$ when $j = n$: the recursive call (line 9) then is $uNecklace(n+1, p, u)$ and everything hinges on whether $n \bmod p = 0$ or not. But the latter check can already be done *before* probing (in $O(n^2)$ time, recall) whether $X[j-p]$ actually is a suitable value for $X[n]$. For any other tried value $i > X[j-p]$ for $X[n]$, the recursive call (line 9) is $uNecklace(n+1, n, \max(i, u))$ and we then know that $n \bmod n = 0$. Hence the test in line 15, as well as lines 19 and 20.

Internal Probing. The leaf probing discussed so far assumes that line 18 is replaced by **return true**. This is unsatisfactory, as no pruning (other than via the p and u parameters) takes place at the internal nodes of the search tree, so that many more leaves are generated than necessary (recall the difference between columns 5 and 2 in Table 1). In the spirit of constraint programming, we ought to perform more pruning when $j < n$. The idea is the same as for leaves (where $j = n$) except that only a strict prefix $X[1, \ldots, j]$ of the circular sequence X is known, so that we can only check whether the minimal renaming of every suffix of $X[1, \ldots, j]$ is lexicographically larger than or equal to $X[1, \ldots, j]$. For example, when searching for a ternary 6-bead unlabelled necklace, assume we have already constructed the pre-necklace 010 and $probe(4, 2, 4)$ is now called to check whether at position $j = 4 < 6 = n$ the variable $X[4]$ can be

assigned the (so far unused) value $i = 2 = u + 1 = k - 1$ under period $p = 4$, so the following comparisons must be made:

$$\begin{aligned}
\underline{2} = \quad & 0 \geq_{\text{lex}} 0 && (4)\\
\underline{02} = \quad & 01 \geq_{\text{lex}} 01 && (3)\\
\underline{102} = \quad & 012 \geq_{\text{lex}} 010 && (2)\\
\underline{0102} = 0102 & \geq_{\text{lex}} 0102 && (1)
\end{aligned}$$

The first and last comparisons will always succeed and can be omitted. Exactly $j - 2$ such renamings and comparisons of tuples of length $O(j-1)$ are thus to be done, hence this internal probing also takes $O(n^2)$ time at worst, since $j = O(n)$. The algorithmic details are straightforward, so we just write a specification into line 18. Again, lazy evaluation of the conjunction should be made. Also, experiments have revealed that failure is detected earlier on average if the starting positions of the suffixes recede from right to left across $X[1, \ldots, j]$, as in the top-down order of the sample comparisons above.

To assess the impact of internal probing, consider again Table 1: column 4 counts the objects when internal probing is on but leaf probing is off (much lower than in column 5); column 3 counts the objects when internal probing is on, leaf probing is off, and the period condition is met; and column 2 counts the unlabelled necklaces, that is the number of objects when internal probing and leaf probing are on. The difference between columns 3 and 2 is the amount of pruning that leaf probing now has to do, and the difference between columns 4 and 3 is the leaf pruning obtained by the period condition. Note that the constant-time period test on the leaves prunes much more than the subsequent quadratic-time leaf probing has to do.

Incremental Internal Probing. Empirically, the internal probing just proposed is on average much more efficient than its $O(n^2)$ worst time suggests, due to the nature of unlabelled necklaces. We now optimise this internal probing into an algorithm taking $O(n)$ time at worst, leading to an enumeration that is systematically faster by a *constant* factor (namely 17% faster in our implementation). The idea is to trade time for space and make the comparisons incremental. Continuing our previous example, having so far constructed the pre-necklace 0102 of a ternary 6-bead unlabelled necklace, $probe(5, 1, 5)$ is eventually called at the next iteration to check whether at position $j = 5 < 6 = n$ the variable $X[5]$ can be assigned the value $i = 1$ under period $p = 5$, so the following comparisons must be made:

$$\begin{aligned}
\underline{\mathbf{1}} = \quad & \mathbf{0} \geq_{\text{lex}} \mathbf{0} && (5')\\
\underline{2\mathbf{1}} = \quad & 0\mathbf{1} \geq_{\text{lex}} 0\mathbf{1} && (4')\\
\underline{02\mathbf{1}} = \quad & 01\mathbf{2} \geq_{\text{lex}} 01\mathbf{0} && (3')\\
\underline{102\mathbf{1}} = \quad & 012\mathbf{0} \geq_{\text{lex}} 010\mathbf{2} && (2')\\
\underline{0102\mathbf{1}} = 0102\mathbf{1} & \geq_{\text{lex}} 0102\mathbf{1} && (1')
\end{aligned}$$

Note that the last four comparisons correspond to the ones given earlier, that the considered suffixes of $X[1, \ldots, j]$ got longer at the *end* by the new (boldfaced) value $i = \mathbf{1}$, and that the minimal renamings of the (non-boldfaced) prefixes remained the *same*. In

other words, only the *scalar* comparisons of the (boldfaced) *last* values matter, since the lexicographic $\geq_{\text{lex}}$ comparisons of the (non-boldfaced) prefixes have already been made until the previous iteration. If the lexicographic comparison until the previous iteration is $=_{\text{lex}}$, as in formulas (1), (3), and (4), then the scalar comparison operator is $\geq$ at the current iteration; if the lexicographic comparison until the previous iteration is $>_{\text{lex}}$, as in formula (2), then *no* scalar comparison need be made at the current iteration. We incrementally maintain a global $k \times n$ matrix m, where $m[i, j]$ gives the minimal renaming of value i if the renaming starts at position j. We also incrementally maintain locally to every search-tree node an n-tuple c of Booleans, where $c[j] = \mathbf{true}$ if the lexicographic comparison from position j until the previous iteration is $=_{\text{lex}}$, that is if the comparison from j is to continue at the current iteration. For example, since the scalar comparison in formula $(3')$ gives $\mathbf{2} > \mathbf{0}$, we set $c[3] \leftarrow \mathbf{false}$ for the next iteration. Using these incremental data structures, the internal probing in line 18 can be replaced by the following specification

$$\mathbf{return} \bigwedge_{2}^{q=j-1} (\mathbf{if}\ c[q]\ \mathbf{then}\ m[i, q] \geq X[j + 1 - q]\ \mathbf{else}\ \mathbf{true})$$

which can be implemented as in Algorithm 4. At most $j-2$ scalar comparisons are to be done, hence this incremental internal probing takes $O(n)$ time at worst, since $j = O(n)$ and the incremental maintenance of $m[i, 1 \ldots j]$ (in lines 1 to 13) and $c[1 \ldots j]$ (in lines 14 to 23) takes $O(n)$ time at worst. Lazy evaluation of the conjunction should be made. Also, experiments have revealed that failure is detected earlier on average if the starting positions of the suffixes recede from right to left across $X[1, \ldots, j]$, as in the top-down order of the sample comparisons above.

Discussion. An analysis of the amortised time complexity of Algorithm 3 is beyond the scope of this paper. Its correctness follows from line 16 capturing the essence of unlabelled necklaces and the correctness of Algorithms 1 and 2:

Theorem 1. *Algorithm 3 correctly enumerates unlabelled necklaces.*

Proof. First assume that $probe$ always returns $\mathbf{true}$. We prove that Algorithm 3 then returns the intersection of the sets of necklaces and unlabelled tuples. ($\subseteq$) Algorithm 3 returns a subset of the set of unlabelled tuples, because line 4 does not systematically return $\mathbf{true}$ (unlike line 4 of Algorithm 1) and because the lowest value to be tried in line 6 is at least (instead of exactly, as in line 6 of Algorithm 1) the lowest available value. Similarly, Algorithm 3 returns a subset of the set of necklaces, because the largest value to be tried in line 6 is at most (instead of exactly, as in line 6 of Algorithm 2) the largest available value. Hence Algorithm 3 returns a subset of the intersection of the sets of necklaces and unlabelled tuples. ($\supseteq$) Conversely, assume a tuple is returned by both Algorithm 1 and Algorithm 2. This means that for each $X[j]$ there was a value i in common in the **try all** statement in line 6 of Algorithm 1 and Algorithm 2, and that the tuple satisfies the test in line 4 of Algorithm 2. Hence the tuple will also be returned by Algorithm 3, because it passes the test in its line 4 and the same values for each $X[j]$ will be tried in the **try all** statement at its line 6.

```
 1: m[i, j] ← 0;
 2: a[0 . . . k − 1] ← false;
 3: s ← 0; q ← j − 1;
 4: while q ≥ 1 ∧ X[q] ≠ i do {scan backwards in X until previous occurrence of i, if any}
 5:    {Invariant: ∀ℓ ∈ D : a[ℓ] ≡ ℓ ∈ X[q + 1 . . . j − 1]}
 6:    {Invariant: s is the number of values distinct from i in X[q + 1 . . . j − 1]}
 7:    if a[X[q]] then
 8:       m[i, q] ← s
 9:    else
10:       s ← s + 1; m[i, q] ← s; a[X[q]] ← true
11:    end if;
12:    q ← q − 1
13: end while; {Assertion: m[i, 1 . . . j] is correctly initialised}
14: c[j] ← true;
15: probe ← true;
16: for q = j − 1 downto 2 do
17:    if m[i, q] > X[j + 1 − q] then
18:       c[q] ← false
19:    else if c[q] ∧ m[i, q] < X[j + 1 − q] then
20:       probe ← false;
21:       break
22:    end if
23: end;
24: return probe
```

Algorithm 4: Incremental internal probing for searching unlabelled necklaces

Now assume there is only leaf probing (i.e., only line 18 of the $probe$ function is replaced by $\mathbf{return\ true}$). By Theorem 2 in Section 3.3 below, line 16 guarantees that only unlabelled necklaces are enumerated among the tuples that are both necklaces and unlabelled necklaces.

Finally, assume there is also probing at internal nodes. The internal probing will then only return $\mathbf{true}$ on branches of the search tree that lead to an unlabelled necklace that has not been seen before. □

To assess the runtime impact of internal probing, consider Table 2: columns 4 and 5 give the enumeration times (in seconds) if there is only leaf probing and also internal probing, respectively. (All experiments in this paper were performed under SICStus Prolog v4.0.3 on a 2.53 GHz Pentium 4 machine with 512 MB running Linux 2.6.24-19.)

Note that we can also wrap the $probe(j, i, p)$ test around lines 7 and 8 of Algorithm 1 only, or of Algorithm 2 only, instead of their "intersection" (Algorithm 3), and still correctly enumerate unlabelled necklaces, since this probing currently exploits neither the semantics of its parameter p nor the fact that it is given a pre-necklace that is an unlabelled tuple. However, the number of unlabelled tuples always exceeds the number of necklaces when n gets sufficiently large (namely $n \geq 7$ for objects over three colours: recall Table 1), so that, in general, it is preferable to start from the intersection of necklaces and unlabelled tuples, since a lot of pruning is then obtained for free.

	necklaces		unlabelled necklaces			
	Algorithm 2	Constraints (3)	Algorithm 3	Algorithm 3	Constraints (1) and (4)	
n	time	time	time (leaf)	time (all)	time	fails
1	0.00	0.00	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0.00	0.01	0
4	0.00	0.00	0.00	0.00	0.01	2
5	0.00	0.00	0.00	0.00	0.01	6
6	0.00	0.01	0.00	0.00	0.02	9
7	0.01	0.01	0.00	0.00	0.05	29
8	0.02	0.02	0.02	0.02	0.14	69
9	0.05	0.04	0.06	0.05	0.40	181
10	0.13	0.11	0.18	0.14	1.14	469
11	0.31	0.25	0.58	0.46	3.78	1240
12	0.90	0.83	1.75	1.42	11.51	3298
13	2.40	2.34	5.64	4.53	34.85	8919
14	6.87	6.24	16.67	13.61	107.80	24329
15	18.60	17.23	52.42	42.36	328.40	66869

Table 2. Enumeration times (in seconds) of objects of length n over 3 colours via dynamic & static (constraint-based) symmetry breaking

3 Static Symmetry Breaking

Consider a CSP $\langle X, D, C\rangle$ where X is a sequence of $n \geq 2$ variables and D is a set of $k \geq 1$ domain values. For simplicity of notation, we assume that $D = \{0, \ldots, k-1\}$; this also has the advantage that the order is obvious whenever we require D to be totally ordered.

3.1 Unlabelled Tuples

To break full value symmetry, it suffices to order the positions of the first occurrences, if any, of each value. Letting $\mathit{firstPos}(i, X)$ denote the first position, if any, of value $0 \leq i < k$ in X under the current assignment, and $n + 1 + i$ otherwise, the following $k - 1$ constraints break full value symmetry [18]:

$$\mathit{firstPos}(0, X) < \mathit{firstPos}(1, X) < \cdots < \mathit{firstPos}(k-1, X)$$

where each $\mathit{firstPos}(i, X) < \mathit{firstPos}(j, X)$ is encoded in [18] by the global constraint $\mathit{intValuePrecede}(i, j, X)$, for which domain consistency can be achieved. A more efficient filtering algorithm can be designed for the entire conjunction of these global constraints, giving the following global constraint [18, 2] (called $\mathit{precedence}$ in [32]):

$$\mathit{intValuePrecedeChain}(D, X) \tag{1}$$

meaning that the order of any two values in the value sequence D is respected in the decision variable sequence X. Unlike the original constraint, in the context of this paper

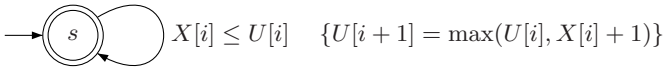


Fig. 1. DFA checker for $intValuePrecedeChain(D, X)$, where decision variable $U[i] \in D \cup \{k\}$ is the smallest unused value after looking up $X[1 \dots i-1]$

we have that the value sequence to be respected is the *entire* totally ordered domain D, so that we need not disregard any values not in D; generalising the following observations to a value sequence strictly included in D is straightforward.

A ground checker for this global constraint can be specified as the deterministic finite automaton (DFA) of Figure 1, so that we get a filtering algorithm using the $automaton$ global constraint [1]. The idea is to create a sequence U of $n+1$ additional decision variables in $D \cup \{k\}$, so that $U[i]$ is the smallest unused value after looking up $X[1 \dots i-1]$, with $U[1] = 0$. As long as $X[i] \leq U[i]$, for i running from 1 to n, we stay in the start state s, which is also an accepting state. If $X[i] > U[i]$ for some i, then we move to an (undrawn) failure state and stay there for any relationship between $X[i]$ and $U[i]$ until $i = n$. The constraint in curly braces ($\{\dots\}$) defines U across the transitions. (Note that the present version of SICStus Prolog does not support counter arrays for $automaton$, so that U cannot be defined explicitly in the transitions as depicted here, but only in conjunction with the $X[i] \leq U[i]$ constraints.) Since the constraint hypergraph corresponding to this DFA is not Berge-acyclic (because each $X[i] \leq U[i]$ constraint shares more than one variable with the corresponding $U[i+1] = \max(U[i], X[i]+1)$ constraint), we are not guaranteed that domain consistency is achieved, but we can enforce domain consistency on the conjunction of these two constraints, using either the $table$ constraint (yielding a DFA that directly corresponds to the encoding in [32]) or the problem-specific DFA checker with $|D|+1$ states in the February 2008 on-line edition of the *Global Constraint Catalogue* [2].

3.2 Necklaces

To break rotation variable symmetry, we apply the so-called *lex-leader* scheme [6], which says that any variant of a wanted solution under all the symmetries of the considered symmetry group must be lexicographically larger than or equal to (a flattening into a linear sequence of) that solution. For necklaces, this means that all the (non-identity) rotations of the sequence X must be lexicographically larger than or equal to X itself [32]:

$$\bigwedge_{q=2}^{n} X[q, \dots, n, 1, \dots, q-1] \geq_{\text{lex}} X[1, \dots, n] \tag{2}$$

These $n-1$ constraints over sequences of *exactly* n elements have been logically minimised in [15] to the following $n-1$ constraints over sequences of *at most* $n-1$ elements:

$$\bigwedge_{q=2}^{n} X[q, \dots, (2q-3) \bmod n + 1] \geq_{\text{lex}} X[1, \dots, q-1] \tag{3}$$

Reading from right to left, this constrains the first $q-1$ elements of X to be lexicographically smaller than or equal to the cyclically next $q-1$ elements of X, for $2 \leq q \leq n$. The DFA for the $\geq_{\text{lex}}$ constraint in [3, 1, 2] is historically the first automaton from which a (domain consistent) propagator was derived.

Note however that up to n decision variables are shared here between the two arguments of each $\geq_{\text{lex}}$ constraint, so that the constraint hypergraph corresponding to this DFA is not Berge acyclic here: even though it is known how to achieve domain consistency in the presence of such variable aliasing [3], this is not implemented in the built-in lex_chain constraint of the current version of SICStus Prolog.

Generalising this to *partial* rotational variable interchangeability, where X is partitioned into subsets with rotational interchangeability, is straightforward.

Future work includes designing a more efficient filtering algorithm for the entire conjunction (3) of global lexicographic constraints, giving a new global constraint, which we propose to call $lexAllRot(X)$.

3.3 Unlabelled Necklaces

The conjunction of the constraints (1) and (3) accepts all necklaces that are unlabelled tuples (just like Algorithm 3 without probing), and therefore accepts a superset of the unlabelled necklaces [32]. For example, the binary necklace 011 is also an unlabelled tuple, but not an unlabelled necklace, because it can be transformed (by rotation and minimal renaming) into the unlabelled necklace 001.

In fact, the rotation variable symmetry and full value symmetry can be broken by the conjunction of constraint (1) with the probing tests in line 16 of Algorithm 3 seen as constraints, ensuring that the minimal renaming of every (non-identity) rotation of the sequence X is lexicographically larger than or equal to X itself:

$$\bigwedge_{q=2}^{n} \underline{X[q,\ldots,n,1,\ldots,q-1]} \geq_{\text{lex}} X[1,\ldots,n] \tag{4}$$

Note that (4) by itself does not suffice, as it accepts the ternary necklace 002, which is not unlabelled: we can only drop (1) if we relax the lower bound on q in (4) from 2 to 1. However, on $q=1$, we have that (4) is logically equivalent to (1), because (4) is then violated if and only if $\underline{X[i]} < X[i]$ at some position i, which means that $X[i] > \max(X[1\ldots i-1])+1$ and hence that (1) is also violated, as X is not an unlabelled tuple because the value of $X[i]$ occurs in X before the value $X[i]-1$. Domain consistency of (4) on $q=1$ is cheaper to achieve on the simpler formulation (1), and this has been confirmed by experiments. The difference of (4) with (2) and (3) lies in the minimal renaming of the left-hand side. The logic minimisation of (2) into (3) does not apply to (4), for the same reason, but such a logic minimisation should be attempted.

We now establish the correctness and completeness of the introduced symmetry-breaking constraints, using (4) for $q \in \{1,\ldots,n\}$ for simplicity of argument. Since the proof is independent of the symmetry group on the decision variables (the cyclic group here), we state the result in generalised form:

Theorem 2. *Given a CSP $\langle X, D, C\rangle$with full symmetry on the values D and a symmetry group G acting on the indices of the n variables X, the constraints:*

$$\bigwedge_{\pi \in G} \underline{X[\pi(1), \ldots, \pi(n)]} \geq_{\text{lex}} X[1, \ldots, n] \tag{5}$$

break all variable and value symmetry.

Proof. The proof is in two stages: first, we show that given an assignment $X[1, \ldots, n] = [d_1, \ldots, d_n]$ there exists a symmetric assignment $[\sigma(d_{\pi(1)}), \ldots, \sigma(d_{\pi(n)})]$ that satisfies (5) for some π in G and some bijection on values σ in $D \rightarrow D$; second, we show that if two symmetric assignments satisfy (5) then they are equal.

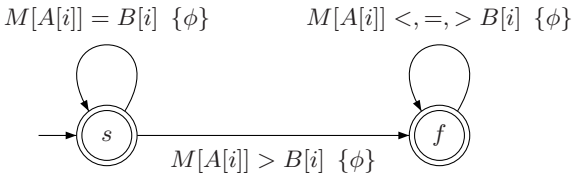
First, given $X[1, \ldots, n] = [d_1, \ldots, d_n]$, consider all permutations $[d_{\pi(1)}, \ldots, d_{\pi(n)}]$, for all $\pi \in G$, and then further consider the minimal renamings $\underline{[d_{\pi(1)}, \ldots, d_{\pi(n)}]}$ of these permutations: the lexicographically smallest element in this list satisfies (5). Second, since the lexicographic order is a total order there is a unique lexicographically smallest element, hence if two assignments that are symmetrically equivalent both satisfy (5) then they must be equal. □

The lex-leader scheme for breaking variable symmetry [6] was adapted in [20] to a particular case of value symmetry. This was later generalised to arbitrary value symmetry groups in [21, 31]. Further, in [31], a $genLexLeader$ constraint was proposed that breaks arbitrary symmetries acting on both variables and values simultaneously. Applying this in our case would require $|G| \cdot |D|!$ such constraints to guarantee full symmetry breaking, whereas in (5) only $|G|$ constraints need to be posted.

A ground checker for the required $\underline{A} \geq_{\text{lex}} B$ global constraint, called $geqLexMin(A, B)$, can be specified as the DFA of Figure 2, so that we get a filtering algorithm using the $automaton$ global constraint [1]. The idea is to augment the classical DFA for $\geq_{\text{lex}}$ [3, 1, 2] with a sequence U of $n + 1$ additional decision variables in $D \cup \{k\}$, so that $U[i]$ is the smallest unused value after looking up $X[1 \ldots i - 1]$, with $U[1] = 0$, as well as with a minimal-renaming bijection M on D (encoded by an $allDifferent$ constraint). As long as $M[A[i]] = B[i]$, for i running from 1 to n, we stay in the start state s, which is also an accepting state. If $M[A[i]] > B[i]$ for some i, then we move to state t, which is an accepting state, and stay there for any relationship between $M[A[i]]$ and $B[i]$ until $i = n$. If $M[A[i]] < B[i]$ for some i, then we move to an (undrawn) failure state and stay there for any relationship between $M[A[i]]$ and $B[i]$ until $i = n$. The constraint in curly braces ($\{\phi\}$) defines U and M across all these transitions. Since the constraint hypergraph corresponding to this DFA is not Berge-acyclic (because each $M[A[i]] \leq U[i]$ constraint shares more than one variable with the corresponding $U[i + 1] = \max(U[i], X[i] + 1)$ constraint, and because the entire M is shared for every i), we are not guaranteed that domain consistency is achieved.

Note also that all n decision variables are shared here between the two arguments of each $geqLexMin$ constraint, so that the constraint hypergraph corresponding to this DFA is also not Berge acyclic for this variable aliasing reason.

In fact, Algorithm 4 for internal probing while searching for unlabelled necklaces is a starting point for a custom filtering algorithm for the $geqLexMin$ constraint.



where $\phi \equiv M[A[i]] \leq U[i]\ \ \&\ \ allDifferent(M)\ \ \&\ \ U[i+1] = \max(U[i], M[A[i]]+1)$

Fig. 2. DFA checker for $\underline{A} \geq_{\text{lex}} B$, denoted by $geqLexMin(A, B)$, where decision variable $U[i] \in D \cup \{k\}$ is the smallest unused value after looking up $X[1 \ldots i-1]$, and decision variable $M[A[i]] \in D$ represents $\underline{A[i]}$, so that M is a bijection on D

Future work includes designing a more efficient filtering algorithm for the entire conjunction (4) of $geqLexMin$ global constraints, giving a new global constraint, which we propose to call $lexAllMinRot(X)$.

3.4 Discussion

To assess the runtimes (in seconds) of dynamic and static symmetry breaking, consider Table 2 again. Unmentioned numbers of backtracks are zero.

For necklaces, columns 2 and 3 reveal an insignificant advantage of the lexicographic constraints (3), under less than domain consistency, over the constant-amortised-time Algorithm 2.

For unlabelled necklaces, the last three columns reveal a huge advantage of Algorithm 3 over constraints (1) and (4). Interestingly, the runtimes are about the same whether we use domain-consistent propagators for (1) or not.

However, these runtimes were obtained in the absence of any problem-specific constraints, and static symmetry breaking usually performs better than dynamic symmetry breaking in the presence of problem-specific constraints. We address this issue in the next section.

4 Experiments

We now experimentally compare the proposed dynamic and static symmetry-breaking methods on real-life scheduling problems containing an (unlabelled) necklace as a combinatorial sub-structure.

4.1 Example: Rotating Schedules

Many industries and services need to function around the clock. Rotating schedules, such as the one in Figure 3(a) (a real-life example taken from [17]) are a popular way of guaranteeing a maximum of equity to the involved work teams. In our example, there

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	x	x	x	d	d	d	d
2	x	x	e	e	e	x	x
3	d	d	d	x	x	e	e
4	e	e	x	x	n	n	n
5	n	n	n	n	x	x	x

(a) Classical rotating schedule

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	d	d	d	d	x	x	e
2	e	e	x	x	d	d	d
3	x	x	e	e	e	e	x
4	x	n	n	n	n	n	n
5	n	x	x	x	x	x	x

(b) Lex-minimal rotation thereof

Fig. 3. A five-week rotating schedule with uniform workload, and its lexicographically minimal rotation

are day (d), evening (e), and night (n) shifts of work, as well as days off (x). Each team works maximum one shift per day. The scheduling horizon has as many weeks as there are teams. In the first week, team i is assigned to the schedule in row i. For any next week, each team moves down to the next row, while the team on the last row moves up to the first row. Note how this gives almost full equity to the teams, except, for instance, that team 1 does not enjoy the six consecutive days off that the other teams have, but rather three consecutive days off at the beginning of week 1 and another three at the end of week 5. We here assume that the daily workload is uniform. In our example, each day has exactly one team on-duty for each work shift, and hence two teams entirely off-duty; assuming the work shifts average 8h, each employee will work $7 \cdot 3 \cdot 8 = 168$h over the five-week-cycle, or 33.6h per week. Daily workload can be enforced by global cardinality (gcc) constraints [22] on the columns. Further, any number of consecutive workdays must be between two and seven, and any change in work shift can only occur after two to seven days off. This can be enforced by $stretch$ constraints [19] on the table flattened row-wise into a sequence. (A filtering algorithm for the $stretch$ constraint, which is not a built-in of SICStus Prolog, was automatically obtained from a DFA model of a constraint checker using the (built-in) $automaton$ global constraint [1].) We assume that soft constraints and cost functions, such as full weekends off as numerous and well-spaced as possible, are enforced by manual selection among schedules satisfying the hard constraints. In our example, there are two full weekends off, in the optimally spaced rows 2 and 5.

4.2 Necklaces

Under the given assumption (uniform workload) and constraints (gcc and $stretch$), *any* rotating schedule has the symmetries of necklaces, when we view it flattened row-wise into a sequence. For example, the schedule in Figure 3(b) is the lexicographically smallest element of the equivalence class to which the schedule in Figure 3(a) belongs, assuming the values are ordered alphabetically ($d < e < n < x$): the former is obtained from the latter by a cyclic left-shift by three positions. Note that the cyclic x stretch in rows 5 and 1 is now entirely on row 5, and that the two optimally spaced 'weekend' days are now Wednesday and Thursday. In other words, it does not matter from what weekday one names the columns, as one can obtain alternative schedules by rotating the circular sequence: $5 \cdot 7 = 35$ schedules, including the one of Figure 3(a), are summarised by the necklace in Figure 3(b).

	unique	Algorithm 2		Constraints (3)		no symmetry breaking		
instance	solutions	time	fails	time	fails	solutions	time	fails
$1d, 1e, 1n, 1x$	14	0.10	2488	0.04	106	114	0.17	391
$1d, 1e, 1n, 2x$	2274	6.66	228823	3.52	9140	17142	18.68	43448
$2d, 1e, 1n, 2x$	4115	47.71	959970	25.01	69704	51014	143.80	419746
$2d, 2e, 1n, 2x$	4950	194.24	2922846	136.56	408669	64556	697.68	2314796
$2d, 2e, 2n, 2x$	3444	587.19	7526564	549.86	1587888	38484	2315.36	8150876

Table 3. Performance comparison over *all* solutions on necklace schedules

In addition to the classical instances in Figure 3, here denoted $1d, 1e, 1n, 2x$, we ran experiments over other instances, namely those over 4 to 10 weeks where the weekly workload is reasonable (33h to 42h) and there are fewer than 100000 unique solutions. For example, instance $2d, 2e, 1n, 2x$ has the uniform daily workload of 2 teams on the day shift, 2 teams on the evening shift, 1 team on the night shift, and 2 teams off-duty; assuming the work shifts average 8h, each employee will work $7 \cdot 5 \cdot 8 = 280$h over the seven-week-cycle, or 40h per week.

Table 3 gives the runtimes (in seconds) and numbers of backtracks (fails) over *all* solutions. On average, when breaking these symmetries statically, the default variable ordering (trying the leftmost variable) is better than first-fail (trying the leftmost variable with the smallest domain) and most-constrained (trying the leftmost variable with the smallest domain that has the most constraints suspended), with the default bottom-up value ordering, hence the runtimes for static symmetry-breaking are given for the default orderings. Static symmetry-breaking for necklaces, in the presence of the problem-specific constraints, is now a lot faster than dynamic symmetry-breaking.

The reason why we have compared the performance over *all* solutions is as follows. The performance to the *first* solution is approximately the same on all these instances (about 0.01 seconds), whether the symmetries are broken dynamically, statically, or not at all. Hence, for this problem, symmetry breaking is not justified if one is only interested in the first solution, even if symmetric non-solutions are also eliminated in the search for it. However, in general, the time ratio to *all* solutions between symmetry breaking and no symmetry breaking is usually a good indicator of that time ratio to the *first optimal* solution, as branch-and-bound essentially iterates over increasingly better solutions in order to pick the best.

To illustrate this claim, Table 4 gives the runtimes and numbers of backtracks to the *first optimal* solution of a sample cost function, namely the maximum number of full weekends off (considering a weekend to be the sixth and seventh days of the week, to avoid having a more complex cost function when breaking symmetries). Indeed, the performance ratios are quite similar to those observed in Table 3, namely a speed-up by a factor of 2 to 5 when breaking symmetries, dynamically or statically. Note that dynamic symmetry breaking is fastest on the last and largest instance.

instance	maximum weekends off	Algorithm 2 time	Algorithm 2 fails	Constraints (3) time	Constraints (3) fails	no symmetry breaking time	no symmetry breaking fails
$1d, 1e, 1n, 1x$	1	0.06	438	0.02	61	0.08	241
$1d, 1e, 1n, 2x$	2	1.40	17002	1.06	4407	3.79	17183
$2d, 1e, 1n, 2x$	2	10.95	140810	8.10	29013	35.36	150980
$2d, 2e, 1n, 2x$	2	48.35	624706	43.97	152175	179.75	768309
$2d, 2e, 2n, 2x$	2	139.58	1833758	147.88	505841	565.96	2470481

Table 4. Performance comparison to the *first optimal* solution (with the maximum number of full weekends off) on necklace schedules

4.3 Partially Unlabelled Necklaces

Under the uniform workload assumption, *some* rotating schedules even have many of the symmetries of *unlabelled* necklaces. In our instances for 4, 5, and 8 weeks, the constraints do not distinguish between the d, e, n work shifts, so that those values are interchangeable.

To break such *partial* value symmetry dynamically, recalling that x is the largest value here ($k-1$ in general), it here suffices to replace line 6 of Algorithm 3 by

$$\textbf{try all } i \in \{X[j-p], \ldots, \min(u+1, k-2)\} \cup \{k-1\}$$

and to make the minimal renamings $\underline{Y}$ in lines 16 and 18 respect the subsets $D_\ell \subseteq D$ of interchangeable values; in our case $D = \{d, e, n\} \cup \{x\}$. We denote the resulting search procedure by Algorithm 3′.

To break this partial value symmetry statically, an $intValuePrecedeChain(D_\ell, X)$ constraint for each subset D_ℓ hinders propagation: a counterexample for partially unlabelled tuples (without the rotation variable symmetry) is given in [32, Section 5]. However, in our case, we conjecture that this *does* suffice, as D is partitioned into only two blocks, one of which is a singleton, hence:

$$intValuePrecedeChain(\{d, e, n\}, X) \tag{6}$$

Together with an adaptation, denoted (4′), of the constraints (4) where $\underline{Y}$ respects the D_ℓ, we have a static symmetry-breaking method for such partially unlabelled necklace schedules.

Table 5 gives the runtimes (in seconds) and numbers of backtracks (fails) over *all* solutions. Static symmetry breaking, in the presence of the problem-specific constraints, is still a lot slower than dynamic symmetry breaking, and even slower than no symmetry breaking (the rightmost three columns are copied from Table 3 for the reader's convenience). Dynamic symmetry breaking for partially unlabelled necklaces (Algorithm 3′), while faster than no symmetry breaking, is however slower on the last two, larger instances than dynamic symmetry breaking for necklaces (compare with Algorithm 2 in Table 3).

Table 6 gives the runtimes and numbers of backtracks to the *first optimal* solution of a sample cost function, namely the maximum number of full weekends off. Again, and

instance	unique solutions	Algorithm 3′ time	Algorithm 3′ fails	Constraints (6) and (4′) time	Constraints (6) and (4′) fails	no symmetry breaking solutions	no symmetry breaking time	no symmetry breaking fails
$1d, 1e, 1n, 1x$	1	0.09	350	3.13	48	114	0.17	391
$1d, 1e, 1n, 2x$	402	12.19	35969	194.94	2964	17142	18.68	43448
$2d, 2e, 2n, 2x$	274	644.47	1380876	29246.82	313587	38484	2315.36	8150876

Table 5. Performance comparison over *all* solutions on partially unlabelled necklace schedules

instance	maximum weekends off	Algorithm 3′ time	Algorithm 3′ fails	Constraints (6) and (4′) time	Constraints (6) and (4′) fails	no symmetry breaking time	no symmetry breaking fails
$1d, 1e, 1n, 1x$	1	0.04	122	1.30	24	0.08	241
$1d, 1e, 1n, 2x$	2	0.76	3301	29.28	1078	3.79	17183
$2d, 2e, 2n, 2x$	2	147.66	350139	5012.51	91937	565.96	2470481

Table 6. Performance comparison to the *first optimal* solution (with the maximum number of full weekends off) on partially unlabelled necklace schedules

ignoring the poor performance of static symmetry breaking, the performance ratios are quite similar to those observed in Table 5, namely a speed-up by a factor of 2 to 5 when breaking symmetries dynamically rather than not at all (the rightmost two columns are copied from Table 4 for the reader's convenience). Dynamic symmetry breaking for partially unlabelled necklaces (Algorithm 3′) is however slower on the last, largest instance than dynamic symmetry breaking for necklaces (compare with Algorithm 2 in Table 4).

5 Conclusions

By bringing together the fields of combinatorial enumeration and constraint programming, we have extended existing results for dynamically and statically breaking the rotation variable symmetry of necklaces into new symmetry-breaking methods dealing also with the additional full value symmetry of unlabelled necklaces. On an example, we have also shown how to specialise these methods when the value symmetry of unlabelled necklaces is only partial. In the absence of problem-specific constraints, the dynamic symmetry-breaking methods outperform the static ones, narrowly for necklaces but largely for unlabelled necklaces. On a real-life scheduling problem we have shown that, in the presence of problem-specific constraints, the static method becomes faster for necklaces, but not for partially unlabelled necklaces.

Most related work was discussed on-the-fly. Furthermore, one should be aware of existing enumeration algorithms for special cases, such as the constant-amortised-time algorithms for unlabelled binary necklaces [4], or for necklaces with fixed content [27] or forbidden substrings [25]. For instance, under the given assumption (uniform workload) and constraints, rotating schedules are necklaces with fixed content, so the algorithm of [27] should be tried instead of Algorithm 2.

Note the difference between our $lexAllRot(X)$ global constraint and the $allperm(M)$ global constraint [12] on an $m \times n$ matrix M of variables, which enforces that the first

row of M is lexicographically smaller than or equal to all permutations of all other rows of M. The $allperm(M)$ constraint was introduced to aid in the incomplete symmetry breaking of the $lex^2(M)$ global constraint [8], which imposes lexicographic orders on the rows and columns of a matrix M of variables where all rows and all columns are assumed interchangeable. Since n of the $n!$ permutations of any row M_i of M are rotations, it would be interesting to compare, in this helper task, the performance of $allperm(M)$ with the performance of the conjunction $lexAllRot(M_0, M_1) \wedge \ldots \wedge lexAllRot(M_0, M_{m-1})$, assuming a suitable binary variant of $lexAllRot$.

Future work includes the quest for a constant-amortised-time enumeration algorithm for unlabelled k-ary necklaces. The fact that all necklaces can be enumerated faster than all unlabelled necklaces (see Table 2 and compare Tables 3 and 5) indicates that such an algorithm might exist.

Also, the simultaneous consideration of *reflection* symmetries and rotation symmetries gives rise to the dihedral symmetry group on the indices and to combinatorial objects known as (unlabelled) *bracelets*. Logically minimised symmetry-breaking constraints for this group have been identified [15], but efficient enumeration algorithms only exist so far for distinguishable values [26].

Furthermore, rotation symmetries on *multi*-dimensional matrices of variables should be considered.

Finally, since the generator functions for (unlabelled) necklaces are known [13], we can add a test to our search procedures that decides in constant time whether to continue enumerating or not, thereby accelerating any proofs of optimality, for instance.

Acknowledgements.

We were supported by grant IG2001-67 of the Swedish Foundation for International Cooperation in Research and Higher Education (STINT), and by grant 70644501 of the Swedish Research Council (VR). We thank N. Beldiceanu, M. Carlsson, J. Sawada, V. Vajnovszki, and T. Walsh for discussions, as well as the providers of the *On-Line Encyclopedia of Integer Sequences* [29] and the *Combinatorial Object Server* [24] for invaluable research tools.

References

1. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In M. Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 107–122. Springer-Verlag, 2004.
2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005:08, Swedish Institute of Computer Science, November 2005. Dynamic on-line version at `http://www.emn.fr/x-info/sdemasse/gccat/`.
3. M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002:17, Swedish Institute of Computer Science, October 2002.
4. K. Cattell, F. Ruskey, J. Sawada, M. Serra, and C. R. Miers. Fast algorithms to generate necklaces, unlabeled necklaces, and irreducible polynomials over $GF(2)$. *Journal of Algorithms*, 37(2):267–282, 2000. Short version in *Proceedings of SODA'00*.

5. W. Y. C. Chen and J. D. Louck. Necklaces, MSS sequences, and DNA sequences. *Advances in Applied Mathematics*, 18(1):18–32, January 1997.
6. J. M. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Proceedings of KR'96*, pages 148–159. Morgan Kaufmann, 1996.
7. M. C. Er. A fast algorithm for generating set partitions. *The Computer Journal*, 31(3):283–284, 1988.
8. P. Flener, A. M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. Van Hentenryck, editor, *Proceedings of CP'02*, volume 2470 of *LNCS*, pages 462–476. Springer-Verlag, 2002.
9. P. Flener and J. Pearson. Solving necklace constraint problems. In M. Ghallab, editor, *Proceedings of ECAI'08*, pages 520–524. IOS Press, 2008.
10. P. Flener, J. Pearson, M. Sellmann, and P. Van Hentenryck. Static and dynamic structural symmetry breaking. In F. Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 695–699. Springer-Verlag, 2006.
11. P. Flener, J. Pearson, M. Sellmann, P. Van Hentenryck, and M. Ågren. Dynamic structural symmetry breaking for constraint satisfaction problems. *Constraints*, forthcoming. Supersedes Technical Report 2007-032 of Department of Information Technology, Uppsala University, Sweden, at `http://www.it.uu.se/research/reports/2007-032/`.
12. A. M. Frisch, C. Jefferson, and I. Miguel. Constraints for breaking more row and column symmetries. In F. Rossi, editor, *Proceedings of CP'03*, volume 2833 of *LNCS*, pages 318–332. Springer-Verlag, 2003.
13. E. N. Gilbert and J. Riordan. Symmetry types of periodic sequences. *Illinois Journal of Mathematics*, 5:657–665, 1961.
14. S. W. Golomb, B. Gordon, and L. R. Welch. Comma-free codes. *Canadian Journal of Mathematics*, 10(5):202–209, 1958.
15. A. Grayland, I. Miguel, and C. Roney-Dougal. Minimal ordering constraints for some families of variable symmetries. In B. Benhamou, editor, *Proceedings of SymCon'07*, 2007. Available at `http://www.cmi.univ-mrs.fr/~benhamou/symcon07/`.
16. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
17. G. Laporte. The art and science of designing rotating schedules. *Journal of the Operational Research Society*, 50(10):1011–1017, October 1999.
18. Y. Law and J. Lee. Global constraints for integer and set value precedence. In M. Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 362–376. Springer-Verlag, 2004.
19. G. Pesant. A filtering algorithm for the stretch constraint. In T. Walsh, editor, *Proceedings of CP'01*, volume 2239 of *LNCS*, pages 183–195. Springer-Verlag, 2001.
20. K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. Technical Report APES-56-2003, APES Research Group, January 2003. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`.
21. J.-F. Puget. An efficient way of breaking value symmetries. In *Proceedings of AAAI'06*. AAAI Press, 2006.
22. J.-C. Régin. Generalized arc-consistency for global cardinality constraint. In *Proceedings of AAAI'96*, pages 209–215. AAAI Press, 1996.
23. C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In R. L. de Mántaras and L. Saitta, editors, *Proceedings of ECAI'04*, pages 211–215. IOS Press, 2004.
24. F. Ruskey. The combinatorial object server. At `http://theory.cs.uvic.ca/root.html`, 2008.
25. F. Ruskey and J. Sawada. Generating necklaces and strings with forbidden substrings. In *Proceedings of COCOON'00*, volume 1858 of *LNCS*, pages 330–339. Springer-Verlag, 2000.

26. J. Sawada. Generating bracelets in constant amortized time. *SIAM Journal on Computing*, 31(1):259–268, 2001.
27. J. Sawada. A fast algorithm to generate necklaces with fixed content. *Theoretical Computer Science*, 301(1–3):477–489, 2003.
28. M. Sellmann and P. Van Hentenryck. Structural symmetry breaking. In *Proceedings of IJCAI'05*, pages 298–303. IJCAI, 2005.
29. N. J. A. Sloane. The on-line encyclopedia of integer sequences. At `http://www.research.att.com/~njas/sequences/`, 2008.
30. P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In *Proceedings of IJCAI'03*, pages 277–282. Morgan Kaufmann, 2003.
31. T. Walsh. General symmetry breaking constraints. In F. Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 650–664. Springer-Verlag, 2006.
32. T. Walsh. Symmetry breaking using value precedence. In G. Brewka, editor, *Proceedings of ECAI'06*, pages 168–172. IOS Press, 2006.