Solving Necklace Constraint Problems

Pierre Flener and **Justin Pearson** [1]

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.

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). The generating functions for counting (unlabelled) necklaces are given in [6], and the sequences of their counts (for $k \leq 6$) can be found in [16].

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 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 with 3^n.

CSPs with an (unlabelled) necklace as a combinatorial sub-structure are not unusual. For example, Gusfield [9, page 12] states that "circular DNA is common and important. [sample organisms omitted.] Consequently, tools for handling circular strings may someday be of use in those organisms". One such problem is studied in [3]. Necklaces occur in coding theory [7], genetics [7], and music [6], while unlabelled necklaces occur in switching theory [6]. We study a real-life problem with (unlabelled) necklaces in scheduling, different from the one in [8].

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 [2]). 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 [13], say, where a generic CP search procedure is proposed for an arbitrary symmetry group acting on the values; however, except for [15] not much dynamic symmetry breaking seems to have been done for groups acting on the 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 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.

In the following, 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. 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.

2 DYNAMIC SYMMETRY BREAKING

Unlabelled Tuples. If the domain values of D are interchangeable, then we impose a total order on D, and the enumeration algorithm of [5], 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 $X[j]$

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

```
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 try
10: end if
```

Algorithm 1: Search procedure for unlabelled tuples [5]

```
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 try
10: end if
```

Algorithm 2: Search procedure for necklaces [2]

(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.

Necklaces. If the variable sequence X is circular, then the enumeration algorithm of [2], 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 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 necklace enumeration remains elusive.

Unlabelled Necklaces. If the variable sequence X is circular *and* the domain values of D are interchangeable, then a constant-amortised-time enumeration algorithm [2] 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

```
1: procedure uneck(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:          uneck(j+1, if i = X[j−p] then p else j, max(i, u))
10:      end if
11:   end try
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: Probing search procedure for unlabelled necklaces

the CP search procedure $uneck$ in Algorithm 3. The initial call is $X[0] \leftarrow 0; uneck(1, 1, -1)$, 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 **true**, then $uneck$ 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 3-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.

Consider the left half of Table 1, giving the numbers of various combinatorial objects of length n over 3 colours: column 7 counts the unlabelled tuples (sequence A124302 in [16]); column 6 counts the necklaces (fewer than the unlabelled tuples for $n \geq 7$; sequence A1867); column 5 counts the necklaces that are unlabelled tuples, that is the number of pre-necklaces when $probe$ always returns **true**; and column 2 counts the unlabelled necklaces (sequence A2076). The difference between columns 5 and 6 (or 7) shows the gain obtained so far for free by Algorithm 3 over Algorithm 2 (or Algorithm 1), but the difference between columns 5 and 2 shows the amount of 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 col or 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-unit) 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

	seq.	probing			seq.	seq.	necklaces		unlabelled necklaces			
	A2076:	internal + leaf		leaf only	A1867:	A124302:	Algo. 2	Cons. (3)	Algo. 3	Algo. 3	Cons. (1) and (4)	
n	unecks	$n \bmod p = 0$	leaves	leaves	necks	utuples	time	time	time (leaf)	time (all)	time	fails
1	1	1	1	1	3	1	0.00	0.00	0.00	0.00	0.00	0
2	2	2	2	2	6	2	0.00	0.00	0.00	0.00	0.00	0
3	3	4	5	5	11	5	0.00	0.00	0.00	0.00	0.00	0
4	6	8	10	13	24	14	0.00	0.00	0.00	0.00	0.01	2
5	9	15	22	36	51	41	0.00	0.00	0.00	0.00	0.01	6
6	26	34	48	97	130	122	0.00	0.00	0.00	0.00	0.03	9
7	53	80	121	268	315	365	0.01	0.01	0.00	0.01	0.07	29
8	146	196	293	732	834	1094	0.01	0.02	0.02	0.02	0.18	69
9	369	490	744	2017	2195	3281	0.04	0.04	0.06	0.06	0.50	181
10	1002	1267	1920	5552	5934	9842	0.11	0.11	0.20	0.16	1.48	469
11	2685	3357	5104	15371	16107	29525	0.24	0.30	0.63	0.49	4.54	1240
12	7434	8996	13635	42624	44368	88574	0.78	0.81	1.95	1.58	13.33	3298
13	20441	24403	37030	118731	122643	265721	2.12	2.22	6.06	4.65	41.04	8919
14	57046	66886	101354	331664	341802	797162	5.91	6.24	18.82	14.50	122.46	24328
15	159451	184770	279895	929883	956635	2391485	16.54	17.25	58.56	44.89	374.12	66865

Table 1. Numbers of objects of length n over 3 colours, and their enumeration times (in seconds) via dynamic & static (constraint-based) symmetry breaking

$O(n^2)$ time at worst. Note that a successful probe incurs the highest cost. The algorithmic details are trivial, 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 the 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 $uneck(n+1, p, u)$ 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 $uneck(n+1, n, \max(i, u))$ 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:

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

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 trivial, 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 the 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 the left half of Table 1: column 4 gives the new numbers of pre-necklaces (much lower than in column 5), and column 3 counts the pre-necklaces that are accepted by the test on the period p. 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 amount of pruning done by the period test. Note that the constant-time period test prunes much more than the quadratic-time probing.

Incremental Internal Probing. Empirically, on average, the internal probing just proposed is 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:

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

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 (the algorithmic details, including the incremental maintenance of c and m, are omitted for space reasons):

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

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 $c[1 \ldots j]$ and $m[i, 1 \ldots j]$ takes $O(n)$ time at worst. Lazy evaluation of the conjunction should be made. Failure is detected earlier on the 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 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. To assess the runtime impact of internal probing, consider the right half of Table 1: the fourth-last and third-last columns 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 v4.0.2 on a 2.53 GHz Pentium 4 machine with 512 MB running Linux 2.6.20.)

3 STATIC SYMMETRY BREAKING

Unlabelled Tuples. To break full value symmetry, it suffices to order the positions of the first occurrences, if any, of each value. Letting $\mathit{firstPos}(i)$ 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 $k-1$ binary constraints break full value symmetry [11]: $\mathit{firstPos}(0) < \mathit{firstPos}(1) < \cdots < \mathit{firstPos}(k-1)$. A more efficient filtering algorithm can be designed for the conjunction of these constraints, giving a new global constraint, called

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

A checker for this global constraint can be specified as a deterministic finite automaton (DFA) (omitted for space reasons), so that we get a filtering algorithm using the $\mathit{automaton}$ global constraint [1].

Necklaces. To break rotation variable symmetry, we apply the so-called *lex-leader* scheme [4], 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 that solution. For necklaces, this means that all the rotations of the sequence X must be lexicographically larger than or equal to X itself:

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

These $n-1$ constraints over sequences of *exactly* n elements have been logically minimised in [8] to the following $n-1$ constraints over sequences of *at most* $n-1$ elements:

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

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 $2 \leq q \leq n$. Future work includes designing a more efficient filtering algorithm for the conjunction of these global lexicographic constraints.

Unlabelled Necklaces. The conjunction of the constraints (1) and (3) accepts all necklaces that are unlabelled tuples (just like Algorithm 3 without probing). In fact, the rotation variable symmetry and full value symmetry can be broken by the constraints (1) together with the probing tests in line 16 of Algorithm 3 seen as constraints:

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

The difference with (2) and (3) lies in the minimal renaming of the left-hand sides. The logic minimisation of (2) into (3) does not apply to (4). A checker for the required $\underline{A} \geq_{\text{lex}} B$ global constraint can be specified as a DFA (omitted for space reasons), so that we get a filtering algorithm using the $\mathit{automaton}$ global constraint [1]. The idea is to augment the classical DFA for $\geq_{\text{lex}}$ [1] with variables representing the smallest value used so far and the minimal-renaming bijection on D (encoded by an $\mathit{allDifferent}$ constraint).

Discussion. The proof of correctness and completeness of the introduced symmetry-breaking constraints is omitted for space reasons. To assess the runtimes (in seconds) of dynamic and static symmetry breaking, consider the right half of Table 1. Unmentioned numbers of backtracks are zero. For necklaces, columns 8 and 9 reveal a slight advantage of Algorithm 2 over constraints (3). For unlabelled necklaces, the last three columns reveal a huge advantage of Algorithm 3 over constraints (1) and (4). 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 (SB) methods on real-life scheduling problems containing an (unlabelled) necklace as a combinatorial sub-structure.

Example: Rotating Schedules. Many industries and services need to function 24/7. Rotating schedules, such as the one in Figure 1 (a real-life example taken from [10]) are a popular way of guaranteeing a maximum of equity to the involved work teams. In our example, there 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 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 $\mathit{stretch}$ constraints [12] on the table flattened row-wise into a sequence. (A filtering algorithm for the $\mathit{stretch}$ constraint, which is not a built-in of SICStus Prolog, was automatically obtained from a DFA model of a constraint checker using

Week	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

Figure 1. A five-week rotating schedule with uniform workload

instance	unique sol's	Algorithm 2 time	Algorithm 2 fails	Constraints (3) time	Constraints (3) fails	no SB time
$1d, 1e, 1n, 2x$	2274	7	228823	4	9140	21
$2d, 1e, 1n, 2x$	4115	50	959970	26	69704	158
$2d, 2e, 1n, 2x$	4950	199	2922846	147	408669	751
$2d, 2e, 2n, 2x$	3444	603	7526564	558	1587889	2581

Figure 2. Performance comparison on necklace schedules

the (built-in) $automaton$ global constraint [1].) We assume that soft constraints, 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.

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. In addition to the classical instance in Figure 1, here denoted $1d, 1e, 1n, 2x$, we ran experiments over other instances. For example, instance $2d, 2e, 1n, 2x$ has the uniform daily workload of 2 teams each on the day and evening shifts, 1 team on the night shift, and 2 teams off-duty. Figure 2 gives the obtained runtimes (in seconds) and numbers of backtracks (fails) over *all* solutions. The time ratio to *all* solutions between SB and no-SB is a good indicator of that time ratio to the *first optimal* solution (say, with the maximum number of full weekends off), as branch-and-bound essentially iterates over many solutions in order to pick the best. On average, when breaking the 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, in the presence of the problem-specific constraints, is now faster than dynamic symmetry-breaking.

Partially Unlabelled Necklaces. Under the uniform workload assumption, *some* rotating schedules even have many of the symmetries of *unlabelled* necklaces. In our instances for 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, it 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, it suffices to post one $orderedFirstOccurrences(X, D_\ell)$ for each subset D_ℓ:

$$firstPos(d) < firstPos(e) < firstPos(n) \qquad (5)$$

Together with an adaptation, denoted (4′), of constraints (4) where $\underline{Y}$ respects the D_ℓ, we have a static symmetry-breaking method for such partially unlabelled necklaces. Figure 3 gives the obtained 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.

instance	unique sol's	Algorithm 3′ time	Algorithm 3′ fails	Cons. (5) and (4′) time	Cons. (5) and (4′) fails
$1d, 1e, 1n, 2x$	402	13	35969	205	2964
$2d, 2e, 2n, 2x$	274	703	1380876	31193	313587

Figure 3. Comparison on partially unlabelled necklace schedules

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.

One should be aware of existing enumeration algorithms for special cases, such as the constant-amortised-time algorithms for unlabelled binary necklaces [2], or for necklaces with fixed content [14]. For instance, under the given assumption (uniform workload) and constraints, rotating schedules are necklaces with fixed content, so the algorithm of [14] should be tried instead of Algorithm 2.

Future work includes the quest for a constant-amortised-time enumeration algorithm for unlabelled k-ary necklaces.

Acknowledgements. We are supported by grant IG2001-67 of the Swedish Foundation for International Cooperation in Research and Higher Education, and by grant 70644501 of the Swedish Research Council. We thank J. Sawada and V. Vajnovszki for discussions.

REFERENCES

[1] N. Beldiceanu, M. Carlsson, and T. Petit, 'Deriving filtering algorithms from constraint checkers', *CP'04*, *LNCS* **3258**:107–122. Springer.
[2] 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).
[3] W. Y. C. Chen and J. D. Louck, 'Necklaces, MSS sequences, and DNA sequences', *Advances in Applied Mathematics* **18**(1):18–32, (1997).
[4] J. M. Crawford *et al.*, 'Symmetry-breaking predicates for search problems', *KR'96*, pp. 148–159. Morgan Kaufmann, (1996).
[5] M. C. Er, 'A fast algorithm for generating set partitions', *The Computer Journal* **31**(3):283–284, (1988).
[6] E. N. Gilbert and J. Riordan, 'Symmetry types of periodic sequences', *Illinois Journal of Mathematics* **5**:657–665, (1961).
[7] S. W. Golomb, B. Gordon, and L. R. Welch, 'Comma-free codes', *Canadian Journal of Mathematics* **10**(5):202–209, (1958).
[8] A. Grayland, I. Miguel, and C. Roney-Dougal, 'Minimal ordering constraints for some families of variable symmetries', *SymCon'07*, (2007).
[9] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, CUP, 1997.
[10] G. Laporte, 'The art and science of designing rotating schedules', *Journal of the Operational Research Society* **50**(10):1011–1017, (1999).
[11] Y. C. Law and J. Lee, 'Symmetry breaking constraints for value symmetries in constraint satisfaction', *Constraints* **11**(2–3):221–267, (2006).
[12] G. Pesant, 'A filtering algorithm for the stretch constraint', *CP'01*, *LNCS* **2239**:183–195. Springer, (2001).
[13] C. M. Roney-Dougal *et al.*, 'Tractable symmetry breaking using restricted search trees', *ECAI'04*, pp. 211–215. (2004).
[14] J. Sawada, 'A fast algorithm to generate necklaces with fixed content', *Theoretical Computer Science* **301**(1–3):477–489, (2003).
[15] M. Sellmann and P. Van Hentenryck, 'Structural symmetry breaking', *IJCAI'05*, pp. 298–303. IJCAI, (2005).
[16] N. Sloane. The on-line encyclopedia of integer sequences. At `http://www.research.att.com/~njas/sequences/`, 2008.