Generating effective symmetry-breaking predicates for search problems

Ilya Shlyakhter

1 Introduction

An important recent application of SAT solvers has been as backends to formal verification tools [2,3]. In this application, verification questions ("does this algorithm or system have a bug?") are reduced to testing satisfiability of a Boolean formula. Each assignment to a Boolean formula's variables represents some execution of the system, and the formula is constructed so that satisfying instances correspond to erroneous executions.

Many practical systems have groups of symmetric (indistinguishable) components. For example, distributed systems often consist of several nodes running identical code. Symmetric components give rise to isomorphic system runs, related to each other by a relabelling of components according to a symmetry. For example, in a mutual exclusion protocol involving two identical processes and a mutex, for each run that starts with process 0 grabbing the mutex there is an isomorphic run that starts with process 1 grabbing the mutex. In a group of isomorphic runs, either all runs satisfy a correctness property or none do. It is therefore sufficient to examine one run per isomorphism class to determine whether there exists a run violating a property. The challenge is in organizing the search through the space of all runs, so that the search takes advantage of the available symmetries. We will now formalize the problem of searching for an object in a universe divided into isomorphism classes. In the context of formal verification, the universe might contain possible program runs, and we might be searching for a run violating a particular correctness property.

Consider a universe U of combinatorial objects representable by m-bit binary numbers. We will speak interchangeably of an object and its binary representation. Let U be divided into equivalence classes of isomorphic objects. A permutation θ of the m bits is a *symmetry* of the universe iff applying θ to any object $X \in U$ yields an object isomorphic to X. The set of all symmetries is the *symmetry group* of the universe U, denoted by Sym.

For example, n-node digraphs can be represented by $n \times n$ adjacency matrices, and two matrices A, B are isomorphic iff there exists a permutation θ of the n nodes such that $\theta(A) = B$, where $(\theta(A))_{i,j} = A_{\theta(i),\theta(j)}$. Note that θ is a permutation of the n *nodes* of the digraph, but it also acts on the n^2-bit *adjacency matrices*, because each permutation of the nodes induces a corresponding permutation of the adjacency matrix bits. The symmetry group Sym has order $n!$ and is isomorphic to σ_n, the symmetric group of order n.

Suppose you need to find an object X from a universe U, satisfying a property $P(X)$ (or determine that no such object exists). Suppose also that P is preserved under isomorphism, i.e. is constant on each isomorphism class. Enumerating all elements of U and testing P on each is clearly wasteful: it's enough to test P on one object per isomorphism class. For some classes of objects, procedures exist for isomorph-free exhaustive generation [4–6]. Faster generation procedures may be developed at the cost of generating more than one labeled object per isomorphism class and/or repeating objects.

If no object in U satisfies P, the generate-and-test approach must explicitly generate a complete representation of at least one representative per isomorphism class to verify unsatisfiability. On the other hand, backtracking methods [7] can rule out entire sets of objects without explicit generation, by determining that no object extending a *partial* binary representation satisfies P. If

P can be encoded as a polynomial-size Boolean constraint on the bits of the fixed-length binary representation (as opposed to black-box computer code), backtracking methods for satisfiability can be used. Such methods can significantly outperform explicit generate-and-test approaches, as demonstrated by satisfiability encoding of planning problems [8].

Crawford et al [1] have proposed an approach to taking advantage of isomorphism structure in this framework. We define a *symmetry-breaking predicate* on U, $SB(X)$, which is *true* on at least one *representative* object per isomorphism class. We then test for satisfiability of $P'(X) = P(X) \wedge SB(X)$. Since P is constant on each isomorphism class, P' is satisfiable iff P satisfiable. Moreover, P' is solved much faster than P by backtracking, because it is more constrained: the algorithm will backtrack if none of the extensions of its current partial instantiation are isomorphism class representatives selected by SB. Experiments show that symmetry-breaking predicates can reduce search time by orders of magnitude with no changes to the search algorithm [1,9].

The difficulty of this approach lies in generating the symmetry-breaking predicate. In general, generating a *complete* symmetry-breaking predicate (*true* of exactly one representative per isomorphism class) is NP-complete [1]; the practical choice is between *partial* symmetry-breaking predicates, *true* of at least one (typically more than one) representative per isomorphism class. To be effective, the predicate must rule out a large fraction of objects from each isomorphism class. On the other hand, the predicate must be compact; otherwise, checking the predicate's constraints at each search node will slow down the search, erasing the benefit of expanding fewer search nodes. Balancing these contradictory requirements is the subject of this paper.

The rest of the paper is organized as follows. Section 2 summarizes prior approaches and points out their deficiencies. Section 3 describes the generation of symmetry-breaking predicates for several classes of combinatorial objects. Section 4 gives a uniform optimality measure for symmetry-breaking predicates, and evaluates the predicates from Section 3 according to this measure. Section 5 gives some experimental evidence that the predicates described in this paper can improve search time. Section 6 describes directions for future work.

2 Prior work

In his original paper on symmetry-breaking predicates, Crawford proposes the following general framework for predicate generation. Fix an ordering of the bits in the object's binary representation. This induces a strict lexicographical ordering on all objects. Construct a symmetry-breaking predicate which is true on the *lexicographically smallest* object in each isomorphism class, as follows.

Let V be a fixed ordering of the bits of the binary representation. Then

$$\bigwedge_{\Theta \in Sym} V \leq \theta(V)$$

is a symmetry-breaking predicate, true of only the lexicographically smallest object in each symmetry class. This predicate explicitly requires that *any* symmetry map either fix the the representative object, or map it to a lexicographically higher object – i.e. that the representative object be lexicographically smaller than any isomorphic object. (We assume that the symmetries of the problem are known; a symmetry-breaking framework that incorporates detection of available symmetries is described in [**?**].)

Unfortunately, in many important cases Sym is very large. For example, for n-node digraphs $|Sym| = n!$, because any permutation of the graph's nodes (and the corresponding permutation of adjacency matrix entries) leads to an isomorphic graph. Crawford suggests mitigating the problem by replacing Sym with a polynomial-size *subset* $Sym' \in Sym$, thus requiring that the object be lexicographically smallest with respect to only some of the symmetries.

Crawford gives no formal guidance on choosing the subset of symmetries to break or the fixed variable numbering to use. This paper begins to fill the gap by describing polynomial-size symmetry-breaking predicates for some common combinatorial objects. For some objects, we refine Crawford's algorithm by determining Sym' and V. For others, we present new predicate constructions, giving a concrete alternative to Crawford's lexicographic approach. Problem-specific symmetry-breaking predicates for some classes of combinatorial objects have previously been studies in [**?**].

Crawford uses empirical measurements to gauge the effectiveness of his symmetry-breaking predicates. While such end-to-end tests are certainly useful, they give no hint of optimality of a given predicate, and reflect peculiarities of a particular backtracking algorithm (such as the dynamic variable-ordering heuristic [7]) besides the inherent complexity reduction brought by the predicate. We present an alternative approach which directly measures predicate pruning power, and gives an optimality measure relative to a complete symmetry-breaking predicate.

3 Generating symmetry-breaking predicates

In this section, we present methods for generating symmetry-breaking predicates on several classes of combinatorial objects: acyclic digraphs, permutations, direct products, and functions. These objects commonly occur in formal descriptions of system designs [10], the analysis of which motivates this work. Each subsection deals with one class of combinatorial objects, describing the binary representation, the isomorphism classes, and the construction of the symmetry-breaking predicate in terms of the binary representation.

3.1 Acyclic digraphs

Let U be the set of $n \times n$ adjacency matrices representing acyclic digraphs. Two matrices representing isomorphic digraphs are isomorphic. The symmetry group Sym has order $n!$.

Any acyclic digraph has an isomorphic counterpart that is topologically sorted with respect to a given node ordering. In terms of adjacency matrices, this means that every isomorphism class of adjacency matrices representing acyclic digraphs includes an upper-triangular matrix (since the lower triangle represents "backwards" edges from higher-numbered to lower-numbered nodes). Our symmetry-breaking predicate simply requires all entries below the diagonal to be $false$. This does not completely break all symmetries, but as measurements in section 4.1 show, breaks most.

Additionally, this symmetry-breaking predicate, together with the requirement that diagonal entries be $false$ (eliminating self-loops), implies the acyclicity constraint, so no additional constraints on the matrix are needed. By contrast, expressing the acyclicity constraint on general digraphs typically takes a constraint of size $\Omega(MatMult(n) \log n)$, where $MatMult(n)$ is the complexity of matrix multiplication. (The constraint involves computing the transitive closure of the adjacency matrix and asserting that it has no diagonal entries; the transitive closure is computed by using repeated squaring of the adjacency matrix.) Shorter constraints require less time to check at every search node, leading to faster search. In general, in cases where not all binary representations represent valid combinatorial objects from our universe U, constraints restricting the object to valid values are separate from the symmetry-breaking predicate. This example illustrates a new use of symmetry-breaking predicates: to reduce the size of original problem constraints.

Note that this symmetry-breaking predicate does not use Crawford's methodology. It's not even clear that a single fixed variable ordering exists which corresponds to this predicate. The next section on permutations gives another example of a symmetry-breaking predicate not based on lexicographic comparison.

3.2 Permutations

Let U be the set of $n \times n$ binary matrices representing permutations of n items. Matrix A represents the permutation mapping i to j iff $A_{i,j}$ is true. A matrix A represents a valid permutation (is a *permutation matrix*) iff every column and every row has exactly one *true* bit.

Two permutations are isomorphic if they have the same cycle structure, i.e. the same multiset of cycle lengths. Thus, an isomorphism class of permutation matrices corresponds to one permutation on a set of n indistinguishale objects. We define a canonical representative of each isomorphism class, and give a polynomial-size Boolean predicate on permutation matrices which is true only of the canonical representatives. We thus achieve full symmetry-breaking with a polynomial-size predicate.

The canonical form is most easily explained using cycle notation for permutations [11]. We require that each cycle consist of a continuous segment of items, that each item map to the immediately succeeding one or (for highest-numbered item in a cycle) to the smallest item in the cycle, and that longer cycles use higher-numbered items than shorter ones. For example, the permutation (12)(345) is in canonical form, but the isomorphic permutations (123)(45), (12)(354) and (15)(234) are not. Formally, given an $n \times n$ permutation matrix A, we have the following predicate in terms of the Boolean entries $A_{i,j}$:

$$
\begin{aligned}
&(\forall i, j | (j > i + 1) \Rightarrow \neg A_{i,j}) \bigwedge \\
&\quad ((\forall i, j | ((j > i) \wedge A_{j,i}) \Rightarrow
\end{aligned}
$$

$$((\wedge_{k=i..(j-1)} A_{k,k+1}) \bigwedge (\wedge_{k=(j+1)..(2j-i)} \neg A_{k,j}))))$$

In this predicate, the condition $(j > i+1) \Rightarrow \neg A_{i,j}$ requires that an item mapped to a higher-numbered item map to the immediately succeeding item: e.g. 3 must map either to 4 (in which case 3 is not the highest-numbered item in its cycle), or to an item numbered not higher than 3 (in which case 3 *is* the highest-numbered item in its cycle). The condition $\wedge_{k=i..(j-1)} A_{k,k+1}$, implied by a backward edge $A_{j,i} (i < j)$, says that every backward edge implies the corresponding forward cycle: e.g. if 5 maps to 3 then 5 must be the highest-numbered item in the cycle and the cycle must be (345). The condition $\wedge_{k=(j+1)..(2j-i)} \neg A_{k,j}$, implied by the presence of a cycle $(i\ i+1\ \ldots\ j-1\ j)$, requires the immediately succeeding cycle to be no shorter, in effect sorting cycles by increasing length: e.g. the cycle (345) excludes the cycles (6) and (67). Together with the original constraints restricting A to be a permutation matrix, these constraints permit exactly one permutation with a given multiset of cycle lengths, i.e. one permutation from each isomorphism class.

The size of this predicate is $O(n^3)$ (since its second and largest conjunct is generated by three nested loops iterating over n indices). $O(n^3)$ matches the order of growth of the original constraints. It may be possible to reduce this order of growth by introducing auxiliary Boolean variables, but since n is typically small (under 15) in our analyses, cubic growth has been acceptable.

3.3 Relations

Consider the direct product $D = D_1 \times \ldots \times D_k$ of k disjoint finite nonempty sets (we call them *domains*). We define our universe U to be $\mathbb{P}(D)$, the power set of D. Each element of U, called a *relation*, can be represented by $\prod_{i=1}^{k} |D_i|$ bits. Each bit corresponds to an ordered k-tuple $(d_1, \ldots, d_k)$, $d_i \in D_i$, and is *true* in the binary representation of a relation iff the relation contains the corresponding ordered k-tuple. We will speak interchangeably of the bits and corresponding ordered k-tuples.

Isomorphism classes are defined by treating elements within each domain as indistinguishable. The symmetry group Sym of our universe U is isomorphic to direct product of k symmetric groups: $Sym \cong \sigma_{|D_1|} \times \ldots \times \sigma_{|D_k|}$. An element $\Theta = (\theta_1, \ldots, \theta_k)$ of Sym maps a relation r to a relation r', such that r' contains an ordered tuple $(d_1, \ldots, d_k)$ iff r contains the ordered tuple $(\theta_1^{-1}(d_1), \ldots, \theta_k^{-1}(d_k))$.

With $|Sym| = \prod_{i=1}^{k} |D_i|!$, direct application of Crawford's method is impractical: the number of symmetries (and hence the size of Crawford's lex-leader predicate) grows super-exponentially with the sizes of the domains D_i. Nevertheless, it is possible to break all symmetries which permute a *single* domain with a linear-size predicate. Even though such symmetries represent only a tiny fraction of all symmetries, experiments show that this predicate rules out most of the isomorphic objects.

We start with an example for the case $k = 2$, then generalize to arbitrary k.

Consider a binary relation $r \in A \times B$, $A = \{a_0, a_1, a_2\}$, $B = \{b_0, b_1, b_2\}$. Let us use the following orderly numbering V for bits of the binary representation of r:

	b_0	b_1	b_2
a_0	1	2	3
a_1	4	5	6
a_2	7	8	9

Under this numbering, Crawford's symmetry-breaking condition for the symmetry exchanging a_0 with a_1 and fixing all other elements (denoted $a_0 \leftrightarrow a_1$) is

$$\overline{123456789} \leq \overline{456123789}$$

which simplifies to $\overline{123} \leq \overline{456}$. Together with the condition for $a_1 \leftrightarrow a_2$, we have

$$\overline{123} \leq \overline{456} \leq \overline{789}$$

which breaks all symmetries permuting only A. Similarly, the conditions for $b_0 \leftrightarrow b_1$ and $b_1 \leftrightarrow b_2$ together simplify to

$$\overline{147} \leq \overline{258} \leq \overline{369}$$

breaking all symmetries which permute only B. Together, these conditions allow only those relations for which permuting either the rows *or* the columns (but not both simultaneously) leads to a lexicographically higher (or the same) relation, according to the given bit ordering. These conditions still allow values of r mapped to lexicographically lower values by symmetries which permute *both* A and B.

In general, consider a relation $r \in D_1 \times D_2 \times \ldots \times D_k$. We use Crawford's lexicographic method with the following numbering. Denoting the elements of D_i as $a_{i,0}, a_{i,1}, \ldots, a_{i,|D_i|-1}$, we number the bit corresponding to tuple $(a_{1,e_1}, \ldots, a_{k,e_k})$, $0 \leq e_i < |D_i|$, as

$$\sum_{i=1}^{k}(e_i \times \prod_{j=i+1}^{k} |D_j|)$$

Now consider a transposition $\theta = a_{i,p} \leftrightarrow a_{i,p+1}$. The effect of this transposition on the binary representation of r is to fix all k-tuples except those with p or $p+1$ as their i'th coordinate, and among the tuples with p or $p+1$ as their i'th coordinate, to swap k-tuples differing only in their i'th coordinate. Within each pair of swapped tuples, the tuple with $p+1$ in i'th coordinate is numbered *higher* than the tuple with p in i'th coordinate. Therefore, Crawford's $V \leq \theta(V)$ condition reduces to $P \leq P'$, where P lists the bits corresponding to k-tuples with p in i'th coordinate, in increasing order by number in our numbering, and P' lists the bits corresponding to k-tuples with $p+1$ in i'th coordinate, in increasing order by number in the numbering. Then the right-hand side of Crawford's $V \leq \theta(V)$ condition for $a_{i,p} \leftrightarrow a_{i,p+1}$ equals the left-hand side of the condition for $a_{i,p+1} \leftrightarrow a_{i,p+2}$, so asserting the condition for adjacent pairs of elements breaks *all* permutations which permute only D_i.

The size of this predicate, expressed in conjunctive normal form (CNF), is linear in the size of each domain. The size of a single n-bit comparator is O(n) [1]. For each domain D_i, we have $|D_i|$ comparators of length $\prod_{j \in \{1,\ldots,i-1,i+1,\ldots,k\}} |D_j|$ for a total comparator size of $O(k \times \prod_{i=1}^{k} |D_i|)$. Measures of effectiveness of this predicate are given in section 4.2.

The fact that any matrix has a doubly lexical ordering has been shown previously [12]. Relations of arity greater than two were not considered, and the work not related to generation of symmetry-breaking predicates. The use of doubly lexical orderings for symmetry-breaking predicates was reported independently in [13].

3.4 Functions

A function is a restricted kind of relation: a two-dimensional relation $r \in A \times B$ with each element of A (the domain) related to *exactly one* element of B (the range). Two functions are isomorphic iff they have the same multiset of preimage sizes. In analyses of relational specifications [10], functions occur more frequently than general relations. For functions, we give a polynomial-size symmetry-breaking predicate which breaks *all* symmetries.

First, we break all symmetries permuting only A by sorting the rows of r as binary numbers, as in the preceding section. For notational convenience, here we make the leftmost column (the bits corresponding to b_0) the least significant bit. Second, we sort the columns by the count of *true* bits. Formally, the constraints on r read

$$(\forall i \in \{0,\dots,|A|-2\}| \\ (\overline{r_{i,|B|-1}r_{i,|B|-2}\cdots r_{i,1}r_{i,0}} \leq \overline{r_{i+1,|B|-1}r_{i+1,|B|-2}\cdots r_{i+1,1}r_{i+1,0}})) \bigwedge \\ (\forall j \in \{0,\dots,|B|-2\}|(|\{i|r_{i,j}\}| \leq |\{i|r_{i,j+1}\}|))$$

We show that together, these constraints define a *complete* symmetry-breaking predicate.

Since r represents a function, there are $|B|$ possible values for a row of r. Sorting the rows of r makes identical rows adjacent, so that the preimage of each $b_j \in B$ occupies a continous segment of A. In addition, for $i < j$, rows mapped to b_i represent smaller binary numbers than rows mapped to b_j. Therefore, elements of A mapped to $b_j \in B$ have lower indices in A than elements of A mapped to b_{j+1}. Alternatively, listing the elements of A in increasing order by index, we first list the elements that map to b_0 (if any), followed by the elements that map to b_1 (if any), and so on, with the elements that map to $b_{|B|-1}$ (if any) at the end of the list.

We now show that adding the second requirement, that the columns be sorted by cardinality (the count of *true* bits in the column), forces a canonical form. Since all matrices in an isomorphism class have the same multiset of preimage sizes (i.e. column cardinalities), sorting the columns by cardinality uniquely determines the cardinality of each column. In other words, all matrices in an isomorphism class satisfying the column-sorting condition have the same cardinalities in the corresponding columns. But given the constraints described in the preceding paragraph, this uniquely determines the image in B of each $a_i \in A$. If $c_j = |\ \{i|r_{i,j}\}|$, i.e. c_j is the cardinality of th j'th column, then the first c_0 elements of A must map to $b_0 \in B$, the next c_1 elements of A must map to $b_1 \in B$, and so on.

For example, here are three isomorphic function matrices satisfying the row-sorting condition:

$$
\begin{array}{cccccc}
 & b_0 & b_1 & b_2 & b_3 & b_4 \\
a_0 & 1 & 0 & 0 & 0 & 0 \\
a_1 & 1 & 0 & 0 & 0 & 0 \\
a_2 & 1 & 0 & 0 & 0 & 0 \\
a_3 & 0 & 0 & 1 & 0 & 0 \\
a_4 & 0 & 0 & 0 & 0 & 1 \\
a_5 & 0 & 0 & 0 & 0 & 1
\end{array}
\quad
\begin{array}{cccccc}
 & b_0 & b_1 & b_2 & b_3 & b_4 \\
a_0 & 0 & 1 & 0 & 0 & 0 \\
a_1 & 0 & 1 & 0 & 0 & 0 \\
a_2 & 0 & 0 & 1 & 0 & 0 \\
a_3 & 0 & 0 & 1 & 0 & 0 \\
a_4 & 0 & 0 & 1 & 0 & 0 \\
a_5 & 0 & 0 & 0 & 1 & 0
\end{array}
\quad
\begin{array}{cccccc}
 & b_0 & b_1 & b_2 & b_3 & b_4 \\
a_0 & 0 & 0 & 1 & 0 & 0 \\
a_1 & 0 & 0 & 0 & 1 & 0 \\
a_2 & 0 & 0 & 0 & 1 & 0 \\
a_3 & 0 & 0 & 0 & 0 & 1 \\
a_4 & 0 & 0 & 0 & 0 & 1 \\
a_5 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Only the rightmost one also orders the column cardinalities, and is the only matrix in the isomorphism class allowed by our symmetry-breaking predicate.

3.4.1 Size of the constraint

4 Measuring effectiveness of symmetry-breaking predicates

Symmetry-breaking predicates are designed to speed up search, so it would seem natural to judge their effectiveness by measuring the reduction in search time. This approach has several problems, however. Search times can be highly dependent on the particular backtracking algorithm, and on parameter settings such as the splitting heuristic [7]. The addition of the symmetry-breaking predicate changes the whole search tree (since splitting choices are determined by the entire constraint set), so the comparison to the original constraint problem is not completely clean. Machine-dependent effects such as cache locality can also bias the measurements. Most importantly, end-to-end measurements provide no clue to optimality: how much of the reduction afforded by symmetry are we actually utilizing?

Table 1
Values used to measure efficiency of partial symmetry-breaking predicates.

value	formula	meaning
labeled	$\|U\|$	the number of distinct binary representations
unlabeled	from [14,15]	the number of isomorphism classes
allowed	$\|\{X \in U \| SB(X)\}\|$	# of objects allowed by symmetry-breaking predicate
effic	$\frac{labeled-allowed}{labeled-unlabeled}$	percentage of excludable objects actually excluded
slack	$\frac{allowed}{unlabeled}$	maximum possible improvement factor

As an alternative measure of efficiency, we can directly measure the pruning power of a symmetry-breaking predicate by counting the number of objects satisfying the predicate. For a complete symmetry-breaking predicate, this number is the number of isomorphism classes. For a partial symmetry-breaking predicate, this number will be higher; the question is, how much higher? Where the number of isomorphism classes is known, we can obtain a precise measure of optimality of our partial symmetry-breaking predicate by comparing its pruning effect with the maximum possible pruning effect.

Table 1 describes the numbers computed to measure efficiency of partial symmetry-breaking predicates.

The numbers of isomorphism classes are taken from [14–17]. The number of objects allowed by the predicate is computed by generating the corresponding satisfiability instance, and counting its solutions with the RELSAT solution counter [18]. Correctness of the implementation was verified by doing complete symmetry-breaking for several classes of objects by Crawford's explicit lexicographical method method, and checking that the number of allowed instances matches the number of isomorphism classes.

4.1 Acyclic digraphs

Table 2 gives efficiency for DAGs.

Table 2
Acyclic digraphs: symmetry-breaking efficiency.

n	*labeled*	*unlabeled*	*allowed*	*effic*	*slack*
3	25	6	8	89.47%	1.3
4	543	31	64	93.55%	2.1
5	29,281	302	1024	97.51%	3.4
6	3,781,50	5,984	32,768	99.29%	5.5
7	1,138,779,265	243,668	2,097,152	99.84%	8.6

Table 3
Relations: symmtry-breaking efficiency.

n	*labeled*	*unlabeled*	*allowed*	*effic*	*slack*
8	102,528	303	1,057	99.26%	3.5
9	1,327,360	1,119	3,828	99.80%	3.4
10	52,494,848	5,479	38,160	99.94%	7.0
11	1,359,217,664	32,303	228,852	99.99986%	7.0
12	107,509,450,752	251,135	3,970,438	99.99997%	16

4.2 Relations

We compute the results for binary relations. The number of isomorphism classes of non-homogeneous binary relations is not published, but we can use the number of bipartite graphs, which are closely related to non-homogeneous binary relations. Every non-homogeneous binary relation $r : A \rightarrow B$ is a bipartite graph on $|A|+|B|$ nodes, with every edge connecting a node in A with a node in B. All isomorphic relations correspond to the same graph. On the other hand, every bipartite graph of n nodes corresponds to at least one non-homogeneous binary relation $r : A \rightarrow B$, with $|A|+|B| = n$. Isomorphic bipartite graphs may correspond to different relations. This means that the number of non-isomorphic bipartite graphs of n nodes lower-bounds the number of non-isomorphic binary relations $r : A \rightarrow B$ with $|A|+|B| = n$. Therefore we can lower-bound the efficiency of our symmetry-breaking predicate by aggregating over relations whose dimensions sum to n.

4.3 Permutations

Here the symmetry-breaking is complete. the savings can be computed analytically.

4.4 Functions

Here we can compute the savings analytically, since the number of unlabeled functions is the number of integer partitions of the domain into preimages – as long as the range is at least as large as the domain.

Most often, non-homogeneous binary relations represent functions: $r : A \rightarrow B!$. For instance, each sector belongs to an area. For this common special case, we can break *all* symmetries with a polynomial-size predicate, by asserting canonical form. For the domain, we assert the same condition as for general relations: that the rows are sorted. For the range, we assert that the preimages are sorted by size. Note that because domain and range atoms are indistinguishable, a function is completely determined by the integer partition of its domain into preimages. We show that each function isomorphism class has a single canonical-form member. Consider a function in canonical form. Since row permutations don't change preimage sizes, the only column permutations are within same-size preimages. Now. Preimages must be continuous, and in order.

So, consider canonical form. Permuting just the domain has been ruled out; permuting just the range also; so the only permutations we have to worry about are joint permutations. The point is that the order is insignificant (in terms of the target of each atom); so, we could say we first permute the range, and then the domain. If we permute the range and change anything, we will not be able to fix it by permuting the domain. Therefore we have to permute the range and not change anything, i.e. permute just the domain.

Table 4
Improvements in search time from using the symmetry-breaking predicates.

model	nosymm	symm
INS	47s	> 900s
Chord	168s	> 900s
Firewire	145s	> 900s
MutexRing	598s	> 10000s
Synchronizer	494s	582ss

But as the previous section has shown, permuting just the domain has been ruled out. So, no permutations are possible at all.

5 Improvements in search time

This section gives some evidence that the predicates described in this paper actually improve search time. These predicates are used by the Alloy Analyzer [3], a checker for relational models. In Table 4, analysis times with and without symmetry-breaking are listed for several models. The models analyze aspects of systems described in [19–23]. In most cases, symmetry-breaking produces significant reductions in search time. In the Alloy Analyzer, symmetry-breaking is turned on by default and practically never disabled by users.

6 Conclusion and future work

We have presented a uniform method to gauge the effectiveness *and* optimality of symmetry-breaking predicates. The method measures the inherent simplification of the constraint problem, which, unlike running-time measurements, does not depend on the details of a particular backtracking algorithm. The method hinges on our ability to lower-bound the number of isomorphism classes in the universe; these numbers are available for a wide variety of combinatorial objects.

We have also presented specific polynomial-size symmetry-breaking predicates for the types of states commonly occurring in analysis of relational specifications. Measurements show that these predicate exclude over 99% of excludable assignments, and come within an order of magnitude of the optimum. These illustrate the potential usefulness of predicates not derived from Crawford's conditions.

Most interestingly, breaking a random set of symmetries to small depth often leads to surprisingly effective predicates. Formalizing this observation into a formal randomized symmetry-breaking scheme will be a major goal of future work. Various ways to bias the random selection of symmetries will be investigated. For instance, Crawford's condition for a single symmetry Θ excludes $2^{n-|\Theta|}$ assignments, where $|\Theta|$ is the number of cycles in Θ. This suggests biasing selection towards symmetries with fewer cycles. On the other hand, overlap between sets of states excluded by the selected symmetries should be minimized. This work could relate to work on probabilistic isomorphism testing.

For states containing multiple components, we need to choose component numbering. Knowing the efficiency could be used to select the best alternative (e.g. by looking up the resulting reduction factors for commonly occurring scopes).

Currently, symmetry-breaking predicates are built based on the inherent symmetries of the problem but without regard to the actual property being tested. This forces the predicate to be true for at least one member of each isomorphism class – even for isomorphism classes ruled out by the property being tested. Future work will explore property-specific symmetry-breaking predicates that use the property being tested to yield a more compact predicate.

References

[1] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Fifth International Conference on Principles of*

Knowledge Representation and Reasoning, 1996.

[2] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[3] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01), Vienna*, September 2001.

[4] B. D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26:306 – 324, 1998.

[5] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems*, 20(2):302–343, March 1998.

[6] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1):41–75, August 1996.

[7] Rina Dechter and Daniel Frost. Backtracking algorithms for constraint satisfaction problems. Technical Report 56, UC-Irvine, 1999.

[8] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 1992. http://portal.research.bell-labs.com/orgs/ssr/people/kautz/papers-ftp/satplan.ps.

[9] David Joslin and Amitabha Roy. Exploiting symmetry in lifted csps. In *AAAI97*, 1997.

[10] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of International Conference on Software Engineering, Limerick, Ireland*, 2000.

[11] Herbert Wilf. East side, west side: an introduction to combinatorial families with maple programming. http://www.cis.upenn.edu/ wilf/eastwest.pdf, 1999.

[12] Anna Lubiw. Doubly lexical ordering of matrices. *SIAM Journal on Computing*, 16:854–879, 1987.

[13] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. Van Hentenryck, editor, *Proceedings of Eighth International Conference on Principles and Practice of Constraint Programming (CP02)*. Springer-Verlag, 2002.

[14] R.C.Read. *An Atlas of Graphs.* Oxford University Press, 1998.

[15] Neil J. A. Sloane. Sloane's on-line encyclopedia of integer sequences. http://www.research.att.com/ njas/sequences/.

[16] F. Harary and E.M.Palmer. *Graphical Enumeration.* Academic Press, 1973.

[17] Brendan McKay. Personal communication. http://cs.anu.edu/au/people/bdm/nauty/.

[18] R. Bayardo and J. Pehoushek. Counting models using connected components. In *AAAI Proceedings*, 2000.

[19] J. M. T. Romijn. A timed verification of the (iee) 1394 leader election protocol. In S. Gnesi and D. Latella, editors, *Proceedings of the Fourth International (ERCIM) Workshop on Formal Methods for Industrial Critical Systems (FMICS '99)*, pages pages 3–29, 1999.

[20] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of ACM*, 17(11):643 – 644, November 1974.

[21] Tina Nolte. Exploring filesystem synchronization with lightweight modeling and analysis, August 2002. MIT Masters thesis.

[22] Ian Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conferene on application, technologies, architectures, and protocols for computer communications (SIGCOMM 2001).* ACM Press, 2001.

[23] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE 2000), Grenoble, France*, September 2000.