Towards Automatic Dominance Breaking for Constraint Optimization Problems*

Christopher Mears and **Maria Garcia de la Banda**
Faculty of IT, Monash University, Australia
{chris.mears,maria.garciadelabanda}@monash.edu

Abstract

The exploitation of dominance relations in constraint optimization problems can lead to dramatic reductions in search space. We propose an automatic method to detect some of the dominance relations manually identified by Chu and Stuckey for optimization problems, and to construct the associated dominance breaking constraints. Experimental results show that the method is able to find several dominance relations and to generate effective dominance breaking constraints.

1 Introduction

Dominance relations can be seen as a generalization of the well-known symmetry relations for optimization problems. Thus, dominance breaking offers similar or even greater search reduction opportunities than symmetry breaking. Interestingly, while there are well-known methods to identify and exploit most kinds of symmetry relations, the identification and exploitation of dominance relations has followed a problem-specific treatment (e.g. [Getoor *et al.*, 1997; Prestwich and Beck, 2004; Proll and Smith, 1997]) with little insight into possible generalizations.

Recently, [Chu and Stuckey, 2012] proposed a generic method for manually identifying a large class of dominance relations and exploiting them by manually generating and adding appropriate dominance breaking constraints. Intuitively, their dominance relations are mappings σ that, under a certain condition $cond$, are known to map solutions to better solutions. That is, if $cond$ holds, $\sigma(\theta)$ maps solution θ to another solution, whose objective function value $f(\sigma(\theta))$ is better (smaller/greater) than that of $f(\theta)$. For such mappings, the negation of $cond$ can be used as a dominance breaking constraint. Chu and Stuckey noted that a dominance mapping is good if its associated dominance breaking constraint $\neg cond$ is simple, as simpler constraints will propagate faster and prune more. Since symmetries always map solutions to solutions, their associated $cond$ is relatively simple: it must only ensure that $f(\sigma(\theta))$ is better.

*This research was sponsored by the Australian Research Council grant DP110102258.

We increase the usefulness of Chu and Stuckey's work by automating it, that is, by developing a method to (a) automatically identify symmetries for a given problem, and (b) automatically construct the associated dominance breaking constraints. Note that these dominance breaking constraints are not symmetry breaking constraints, as the key element is for $f(\sigma(\theta))$ to be better. Further, the symmetries need to be detected for the inherent satisfaction problem — that is, the problem without the objective function — or, otherwise, $f(\sigma(\theta))$ will be equal to $f(\theta)$, not better.

Section 3 presents an automatic method focused on ensuring symmetry inference is done very efficiently, as it needs to be done every time we search for a solution to the problem. While the experiments show clear speedups, such efficient inference methods are often unable to infer useful symmetries. Thus, Section 4 extends this approach to use accurate symmetry inference methods for the problem *model*, that is, for a parameterized version of the problem specification that can later be instantiated with particular input data for the parameters. While model-based methods can take more time, they are scalable and, once applied, the dominance breaking constraints generated can be used for all instances of the model, thus compensating for their extra time.

Examining the examples by Chu and Stuckey, we determined that some of their manually identified mappings correspond to *almost symmetries* [Martin, 2005], that is, to mappings that are symmetries of the problem only if a few constraints are ignored. To take advantage of this insight, Section 4 further extends the automatic detection process of Section 3 to infer dominance constraints derived from almost symmetries. In particular, and given the importance of obtaining simple dominance constraints, Section 3 focuses on the class of almost symmetries obtained from eliminating a single constraint from the problem. Our experiments show that the generated dominance breaking constraints can yield significant improvements in search performance.

2 Background

This section provides the necessary background following mainly [Chu and Stuckey, 2012]. Let $\equiv$ denote syntactical identity. A variable v is a mathematical quantity capable of assuming any value from its domain. Given a set of variables V, let Θ_V denote the set of valuations over V where each variable in V is assigned to a value in its domain. A con-

straint c over V is defined by a set of satisfying valuations $sols(c) \in \Theta_V$. Given a valuation θ over $V' \supset V$, we say θ satisfies c if the restriction of θ to V is in $sols(c)$. A domain D over V is a set of unary constraints, one for each variable in V. In an abuse of notation, if A refers to a set of constraints $\{c_1, \ldots c_n\}$, we will also use A to refer to the constraint $c_1 \wedge \ldots \wedge c_n$.

A Constraint Satisfaction Problem (CSP) is a tuple $P \equiv (V, D, C)$, where V is a set of variables, D is a domain over V, and C is a set of constraints, each $c \in C$ defined over a subset of V. A valuation θ over V is a solution of P if it satisfies every constraint in D and C. A Constraint Optimization Problem (COP) $P \equiv (V, D, C, f)$ extends a CSP by adding an objective function f mapping valuations to an ordered set, e.g., $\mathbb{Z}$ or $\mathbb{R}$, with the aim of finding a solution to the CSP that minimises/maximises f. As for [Chu and Stuckey, 2012], we deal with finite domain problems only, where the initial domain D constrains each variable to take values from a finite set of values. Further, for brevity, all objective functions are assumed to be minimised, and we consider CSPs as COPs where $f(\theta) = 0$ for any valuation θ.

In our examples we represent CSPs and COPs using the MiniZinc [Nethercote *et al.*, 2007] modelling language and follow [Mears *et al.*, 2008] in distinguishing between a CSP (or COP) *model* and its *instance*. A model is any specification that can be expressed as a MiniZinc program with at least one unspecified parameter. An instance of a model (the actual CSP or COP) is then defined as the result of extending the model by providing values to all its parameters. While an instance corresponds to a MiniZinc program without unspecified parameters, instances are usually compiled into FlatZinc, a lower level description language with a very restricted constraint format (see Section 3.2).

Dominance relations are defined as follows.

Definition 1. (from [Chu and Stuckey, 2012]) A *dominance relation* $\prec$ for a COP $P \equiv (X, D, C, f)$ is a transitive and irreflexive binary relation on Θ_X such that if $\theta_1 \prec \theta_2$, then either: 1) θ_1 is a solution and θ_2 is a non-solution, or 2) they are both solutions or both non-solutions and $f(\theta_1) \leq f(\theta_2)$.

If $\theta_1 \prec \theta_2$, then θ_1 is said to dominate θ_2. Chu and Stuckey show that any dominated valuation can be safely discarded. Since this can be extended to search nodes in the obvious way, we can also discard dominated search nodes.

Definition 2. (from [Chu and Stuckey, 2012]) Let D_1 and D_2 be the domains from two different search nodes. We say $D_1 \prec D_2$ if $\forall \theta_2 \in sols(D_2), \exists \theta_1 \in sols(D_1)$ s.t. $\theta_1 \prec \theta_2$.

Chu and Stuckey provide a manual method for identifying dominance relations in a COP and adding constraints to exploit them. Their method has four steps:

1. Find mappings $\sigma : \Theta_X \to \Theta_X$ that are likely to map solutions to *better* solutions, that is, $\sigma(\theta) \prec \theta$ often holds.
2. For each σ, find a constraint $scond(\sigma)$ s.t. if $\theta \in sols(C \wedge D \wedge scond(\sigma))$, then $\sigma(\theta) \in sols(C \wedge D)$, i.e., a constraint under which σ maps solutions to solutions.
3. For each σ, find a constraint $ocond(\sigma)$ s.t. if $\theta \in sols(C \wedge D \wedge ocond(\sigma))$, then $f(\sigma(\theta)) < f(\theta)$, i.e., a constraint under which σ maps solutions to better valuations.
4. For each such σ, post the dominance breaking constraint $\neg(scond(\sigma) \wedge ocond(\sigma))$.

A *solution symmetry* (or simply *symmetry*) of CSP $P \equiv (V, D, C)$ is a permutation of the variable/value pairs in $V \times D$ that maps solutions of P to solutions of P and non-solutions to non-solutions [Cohen *et al.*, 2006]. A special case of solution symmetry, called a *valuation symmetry* in [Martin, 2005] (and syntactic symmetry in [Meseguer and Torras, 2001]), occurs when σ maps all the variable/value pairs of each variable to the variable/value pairs of another variable. Importantly, such symmetries always map a valuation to another valuation and, thus, can easily be extended to map constraints to constraints, since the mapping on variables is uniform. This is critical for us, as our method will need to apply symmetries to constraints to obtain other constraints. Many common symmetries (such as variable symmetries) are valuation symmetries.

A permutation group is a set of permutations that is closed under composition and inverse. The solution (and valuation) symmetries of a CSP P form a permutation group, where each element is a permutation on the set of variable/value pairs of P. Given permutations $\{g_1, g_2, \ldots, g_n\}$ for P, if the their closure under composition and inverse is equal to permutation group G, we say that $\{g_1, g_2, \ldots, g_n\}$ *generates* G, and $\{g_1, g_2, \ldots, g_n\}$ is a *generating set* of G.

3 Automatic generation for instances

Given a COP instance I, our method takes the following steps to generate dominance breaking constraints:

1. Extract from I its CSP core $ISat$ by eliminating its objective function f and any variables and constraints that only appear in I to define f,
2. Find the symmetries of $ISat$,
3. For each symmetry σ, generate constraint $f \leq \sigma(f)$.

The remainder of this section details each of these steps.

3.1 Extracting a CSP from a COP instance

To be useful in dominance breaking, a symmetry σ must apply to the "satisfaction core" of the COP instance, rather than to the COP instance itself. Otherwise, $f(\sigma(\theta))$ will be known to be equal to — rather than better than — $f(\theta)$ for any solution θ. Chu and Stuckey assumed a specification of their COP problem $P \equiv (X, D, C, f)$ where everything regarding the objective function is tucked away in f, since it is then easy to obtain the satisfaction part: (X, D, C). In practice, however, this is often not the case.

Example 1. Consider the following COP model of the photo placement problem from the MiniZinc distribution, where each person's name is assigned a different position in such a way that their combined positional preferences (who they would like to be next to) are maximized (equivalently, their negation minimized):

```
1  int: names; int: n_prefs;
2  array[1..n_prefs,0..1] of int: prefs;
3  array[0..names-1] of var 0..names-1:pos;
4
5  constraint alldifferent(pos);
6
7  var 0..n_prefs: satisfies;
8  array[1..n_prefs] of var bool: ful;
9  constraint forall (i in 1..n_prefs) (
10   let { int: pa = prefs[i,0],
11         int: pb = prefs[i,1] }
12   in ful[i] = (pos[pb]-pos[pa] == 1 xor
13                pos[pa]-pos[pb] == 1) );
14 constraint satisfies =
15  sum(i in 1..n_prefs)(bool2int(ful[i]));
16
17 solve minimize -satisfies;
```

Lines 1 and 2 show the model parameters (number of names and of preference pairs, and the array of preferences). Line 3 shows the array of decision variables assigning each position to the person's name. Line 5 shows the only true constraint: an `alldifferent` forcing the names assigned to each position to be different. The variables and constraints introduced from lines 7 to 15 serve only to define the objective variable `satisfies`, since they do not exclude any assignments to the decision variables in array `pos`. When inferring the symmetries of the satisfaction problem, we do *not* want to use these constraints, as they can eliminate some of the symmetries available, particularly if these symmetries are likely to find better solutions.□

Our method obtains the satisfaction part $ISat$ of a COP instance I equivalent to (X, D, C, f), by removing constraints that serve only to define f without reducing the number of solutions in I. To achieve this our method first removes the objective function from the instance. Then, it iteratively removes variables and constraints as follows: If variable y is used in only one constraint c, and c *describes* y, remove c and y.

Definition 3. A constraint c over variables $\{x_1, \ldots, x_n, y\}$ *describes* variable y if the set of solutions $sols(c)$ projected over $\{x_1, \ldots, x_n\}$ is $D(x_1) \times \ldots \times D(x_n)$.

Intuitively, this occurs whenever the solutions to c place no restriction on the values of $x_1, \ldots, x_n$. In practice, this often corresponds to the case where y functionally depends on $x_1, \ldots, x_n$. Clearly, whether a constraint describes a variable or not depends on the domains of its variables.

Example 2. Consider the constraint $x + y = z$, with $D(x) = D(y) = \{0, 1\}$ and $D(z) = \{0, 1, 2\}$. This constraint only describes variable z. However, if the domains were instead $D(x) = \{0, 1, 2\}$, $D(y) = \{0, 1\}$ and $D(z) = \{1, 2\}$, then the constraint would describe variable x instead. □

The MiniZinc-to-FlatZinc compiler detects many cases where a constraint describes a variable. For the remaining constraints, we examine the domains of their variables.

Example 3. Consider again the model introduced in Example 1, instantiated with a data file. After our method eliminates the objective from the associated FlatZinc instance file and iteratively removes variables and constraints as described above, the $ISat$ contains only the decision variables in array `pos` and the `alldifferent` constraint. □

In the worst case, the CSP extraction traverses N times the instance I, where N is the number of constraints in I. As each traversal is linear in the size of I, the overall complexity is quadratic in the size of I.

3.2 Instance-based Symmetry Detection

Once we have $ISat$, we need to find its symmetries. Crucially, the symmetry inference method used must be fast, as it will add to the solving time for the original COP. Complete methods (e.g. [Cohen *et al.*, 2006]) tend to be computationally expensive, even for small instances. Incomplete methods (e.g., [Puget, 2005; Ramani and Markov, 2004; Cohen *et al.*, 2006; Mears *et al.*, 2009]) can be very accurate but slow, as the more accurate ones build a graph G with a vertex per variable/value pair in the instance, such that each automorphism of G corresponds to a solution symmetry.

We formalise and implement the method by [Puget, 2005], obtaining a fast and reasonably accurate method for detecting variable symmetries in a FlatZinc instance file. Space limitations prevent us from describing the method in full; briefly, it constructs a coloured graph (V, E, c) with nodes for every variable, constant and constraint in the problem, in such a way that its automorphisms are symmetries of the instance.

3.3 Automatically Generating Dominance Breaking Constraints

Each of the symmetries found in the previous step is a mapping σ that can be used for dominance breaking. As mentioned before, the general form of a dominance breaking constraint is $scond(\sigma) \rightarrow \neg ocond(\sigma)$. For the case of the mapping σ being a (valuation) symmetry, $scond(\sigma)$ is trivially true and $ocond(\sigma)$ is $\sigma(f) < f$.

Given a COP instance I represented by (X, D, C, f) with satisfaction core $ISat$, our method to automatically generate the dominance breaking constraints from a mapping σ induced from a variable symmetry of $ISat$, is as follows. We first create a renaming ρ that maps every variable x in I to a new fresh variable x', and we extend σ to map every variable not already in σ to itself. Then, we add every new fresh variable x' to I with the same domain as x, we add a constraint equating every variable x in the satisfaction part $ISat$ to $\rho(\sigma(x))$, and for every constraint c in I, we also add $\rho(c)$ unless it is known to be equivalent to a constraint already in I and, thus, redundant. Some constraints can be easily detected as redundant, e.g., if c only constrains variables from the satisfaction part and none in the domain of σ, then $\rho(c)$ is known to be redundant (as it simply copies c using fresh new variables to which the original ones are now equated). Even if c contains variables in σ, the mapping might not affect the semantics of the constraints, e.g., in constraint `alldifferent(pos)`, any mapping that permutes the variables in array `pos` will result in a copy that is equivalent and thus redundant. Finally, we add the dominance breaking constraint, which is simply $f \leq \rho(f)$.

Example 4. Consider an instance I of the model introduced in Example 1, where parameter `names` is set to

4. The FlatZinc compiler renames the variables in array `pos` to `X_0`, `X_1`, `X_2` and `X_3`. These are the only variables in the satisfaction part $ISat$ of the problem. Any permutation of variables `X_0`, `X_1`, `X_2` and `X_3` is a valuation symmetry of $ISat$. Consider the mapping σ that swaps `X_1` and `X_2`. Our transformation creates a renaming ρ that maps every variable (such as `X_0` and `satisfies`) to fresh new variables (such as `X_0_prime` and `satisfies_prime`). It then adds to I all new prime variables with the same domain as their original ones (e.g., `X_0_prime`, `X_1_prime`, `X_2_prime`, and `X_3_prime` are all added with domain 0..3 and `satisfies_prime` with domain 0..`n_prefs`). Also, it adds equations `X_0=X_0_prime`, `X_1=X_2_prime`, `X_2=X_1_prime` and `X_3=X_3_prime`. Then, for every constraint c, it computes $\rho(c)$ and adds it to I unless it knows it is redundant. Finally, it adds the dominance breaking constraint: `constraint satisfies >= satisfies_prime;`

For this problem our method is able to automatically detect most constraint copies as redundant, as they only involve satisfaction variables not affected by the symmetry (like `X_0`, `X_3` here), or permute the order of satisfaction variables in such a way it does not affect the meaning of the constraint due to its commutative nature: a sum in a linear constraint or an array in the `alldifferent`. For example, `alldifferent([X_0, X_1, X_2, X_3])` becomes `alldifferent([X_0_prime, X_1_prime, X_2_prime, X_3_prime])`. While the equations added to I do not make these two constraints identical (since `X_1` is equated to `X_2_prime` rather than to `X_1_prime`, and the same for `X_2`) they are known to be equivalent. □

Constructing the additional variables and constraints is linear in the size of the instance, while detecting redundancies is quadratic, giving a total complexity of $O(N^2M)$, where N is the size of the instance and M is the number of mappings. While this construction can lead to a significant growth in the model, our method is often able to automatically detect many redundancies.

Importantly, we cannot simply add a dominance breaking constraint for every element of the variable symmetry group inferred by our method. The reasons are twofold: first, the typical size of symmetry groups is so large that to do so would incur an enormous overhead which the reduction in search space would not recoup; second, even if the group is small, some of the associated dominance breaking constraints might by themselves already add significant overheads. We would like to add only those dominance breaking constraints that will propagate efficiently and give good search space reduction. We currently use the generating set of symmetries produced by the symmetry detection method. More work needs to be done to better select the symmetries.

3.4 Evaluation: Knapsack

The knapsack optimization problem selects a subset of objects such that the sum of their weights is within a fixed limit and the sum of their profits is maximised (or their negation minimised). The $ISat$ of the original problem I only contains the weight limit, as the profit only contributes to the objective function. The automatic symmetry detection for $ISat$ finds that any two objects with the same weight can be swapped. Concretely, it finds a set of pairwise swaps of objects as a generating set of the symmetry group for the instance. Using each swap as a mapping, we generate dominance breaking constraints and add them to I.

Instance (size-id)	Without Dominance	With Dominance
knap-50-1	307.3	134.0
knap-50-2	544.9	205.6
knap-50-3	366.8	186.7
knap-50-4	595.0	334.8
knap-50-5	>1800.0	1782.9
knap-50-6	122.2	83.3
knap-50-7	144.9	36.2
knap-50-8	544.3	62.3
knap-50-9	140.5	120.4
knap-50-10	797.9	328.5

Table 1: Knapsack experimental results (in seconds).

Table 1 shows the time in seconds to solve instances with 50 objects, with and without automatically generated dominance breaking constraints (instances are from [Chu and Stuckey, 2012]). These and all other experiments use the MiniZinc 2.0 compiler and Gecode 4.3.3. While the dominance breaking constraints automatically obtained by our method are not as strong as those manually identified by [Chu and Stuckey, 2012] (as their mappings are not derived from symmetries), they achieve a large reduction in the search which clearly pays off in execution time, particularly for the larger instances. Although the speed-up obtained here is not as good as that in [Chu and Stuckey, 2012], this is to be expected since their method is manual and problem-specific while ours is automatic and generic.

4 Using models and almost symmetries

The method described in the previous section is simple, powerful, and can lead to good speed-ups. However, in general it adds a significant overhead, as the three steps (extraction of the CSP, symmetry inference, and generation of dominance constraint per symmetry) are performed for every instance. Further, the need for efficiency requires a relatively simple (and thus inaccurate) symmetry inference method. Finally, as mentioned before, we have discovered that some of the mappings manually identified in [Chu and Stuckey, 2012] are almost symmetries, rather than symmetries.

We thus extend the previous method with two fundamental changes. First, we directly work with the model, since this allows us to use more resource-intensive methods whose results can be used for all instances of the model, thus compensating for the extra time required. Second, we infer not only symmetries but also almost symmetries. Intuitively, this new method starts with a COP model M and proceeds as follows:

1. Generate relaxed models $M_1, \cdots, M_m$ of M by removing some of its constraints (see Section 4.1).
2. Find the symmetries of M and of its relaxed versions – which are almost symmetries of M (see Section 4.2).

3. For a selection of the symmetries found in each of these models, generate a (implicative) dominance breaking constraint, as already described in Section 3.3.
4. Create new models by adding to M the constraints generated in the previous step for each symmetry.

To select symmetries in step 3, we map each of the model symmetry patterns [Mears *et al.*, 2008] to the set of symmetries that will be used as mappings for generating dominance breaking constraints. We are currently automating this task, which is the only manual task left in the method.

4.1 Generating relaxed versions of a COP model

We automatically generate relaxed models of any MiniZinc model M by using its structure, that is, its constraint "items", which typically correspond to the high-level constraints of the problem. In particular, assuming M has m constraint items $c_1, \cdots, c_m$, we generate a set of relaxed models $\{M_1, \ldots, M_m\}$, where M_i is obtained by simply removing constraint item c_i from M.

Example 5. The model for the blackhole solitaire (see Section 4.4) has 4 constraints, denoted A, B, C and D. Four relaxed models can thus be derived from it: a model with only constraints A,B,C; one with A,B,D; one with A,C,D; and one with B,C,D.□

Since any symmetry σ of M_i is an almost symmetry of M, such σ can be used to generate the usual dominance breaking constraint of M: $scond(\sigma) \rightarrow \neg ocond(\sigma)$. The main difference is that, while for a symmetry of M the left hand side would be $true$, for σ it is $\sigma(c)$, where c is the conjunction of constraints removed from M to obtain M_i. This is why we remove a single constraint item (i.e., $c \equiv c_i$), to keep constraint $\sigma(c)$ simple.

4.2 Finding the symmetries of the model

Several methods are available to obtain the symmetries of a model, e.g. [Van Hentenryck *et al.*, 2005; Roy and Pachet, 1998; Mancini and Cadoli, 2005; Mears *et al.*, 2008]. We decided to adapt the method of [Mears *et al.*, 2008], since it has been shown to have good accuracy for a wide range of benchmarks. Given a model M (relaxed or not) and n input data sets $d_1, \ldots, d_n$, we proceed as follows. First, we instantiate M with each input data set to obtain the COP instances $I_1, \ldots, I_n$. Second, for each I_i, we extract the CSP instance $ISat_i$ as described in Section 3.1. Third, we use [Mears *et al.*, 2009] to accurately detect a generating set S_i for the symmetries of each $ISat_i$. Finally, we use the method of [Mears *et al.*, 2008] to infer the generating set of symmetries S of (the satisfaction part of) M from the generating set of symmetries $S_1, \ldots, S_n$ previously inferred for every instance.

4.3 Evaluation: Photo Placement

Let us use the Photo Placement problem to show how our method applies to a model rather than to an instance (next section shows the usefulness of almost symmetries). We have considered two models referred to as photo1 and photo2, respectively: the one shown in Example 1 and the one given by Chu and Stuckey [2012]. After the CSP extraction, both models contain only the position variables and the all-different constraint. Therefore, any permutation of the position variables is a (full) symmetry. For such permutation groups we consider two sets of dominance mappings: (a) for all pairs of positions i and j, swaps i and j, and (b) for all pairs of positions i and j, reverse the subsequence between i and j. For both mappings, the dominance constraint automatically generated for photo2 is:

```
constraint sum(i in 1..n-1)
               (p[x_prime[i],x_prime[i+1]])
         - sum(i in 1..n-1)
               (p[x[i], x[i+1]])
         <= 0; □
```

where the `x` array corresponds to the `pos` array in the photo1 model. The main difference between the models obtained with these two mappings is the extra constraint equating the variables to their `primes`. For example, the equating constraint for the reversal mappings is:

```
array [1..n] of var 1..n : x_prime =
  [ x [ if 1 <= i /\ i <= 2
        then 2 - i + 1
        else i endif ] | i in 1..n ];
```

The resulting dominance breaking constraint is the one manually identified by Chu and Stuckey [2012] for this problem.

Table 2 shows the effect on the search in terms of the total number of search nodes obtained when solving different instances of each of the two models without dominance breaking (column None), and with dominance breaking using swap mappings (Swaps) and reversal mappings (Reversals). The instances are again from [Chu and Stuckey, 2012], where the size represents the number of people in the photo. The two models are tested on different instances because photo1 does not scale well to instances with more than about 10 people. Note that in a branch-and-bound search, extra constraints can *increase* the search if they divert it from finding early solutions that would have imposed constraints on the objective. Clearly, the reversal mappings consistently give a smaller search than the swap mappings for both models. In the first model, the dominance breaking constraints increase the node count, while in the second model they decrease it. A likely explanation for the poor performance in the first model is that the dominance breaking constraints propagate poorly and, thus, the lack of pruning due to the exclusion of early solutions is not compensated by later reductions in the search tree size. We are exploring ways to automatically detect and transform the constraints to improve pruning.

Instance (size-id)	None	Swaps	Reversals
photo1-9-1	35,774	44,537	37,132
photo1-11-2	480,109	809,282	506,268
photo2-12-1	4,015,993	1,199,867	798,781
photo2-12-2	6,405,713	1,574,469	1,102,183
photo2-14-2	227,036,415	35,153,627	17,583,257

Table 2: Node counts for photo placement.

Table 3 shows the effect on running time from the reversal mappings and the considerable reduction in time achieved by automatically excluding redundant constraints. For those instances where the dominance breaking constraints cause a

Instance (size-id)	None	Revs. incl. redundant	Reversals
photo1-9-1	0.147	1.472	1.103
photo1-11-2	1.140	31.220	23.332
photo2-12-1	20.881	43.022	22.140
photo2-12-2	31.126	57.753	28.287
photo2-14-2	1445.584	1394.694	649.471

Table 3: Effect of removing redundant constraints.

Instance (size-id)	None	Reversals
photo2-14-1	1357.6	836.2
photo2-14-2	1445.6	649.5
photo2-14-3	87.7	123.2
photo2-14-4	679.8	395.0
photo2-14-5	1352.6	770.8
photo2-14-6	1359.7	613.2
photo2-14-7	413.5	407.8
photo2-14-8	510.4	445.5
photo2-14-9	1127.0	637.3
photo2-14-10	>1800.0	1139.8

Table 4: Running time for photo placement.

large reduction in search (all instances of photo2), there is also a big reduction in running time. Table 4 shows the reduction in running time over the full set of size 14 instances; in almost all cases the dominance breaking constraints reduce the running time, and often the reduction is significant.

4.4 Evaluation: Blackhole Solitaire

Let us now consider the effect of almost symmetries. The blackhole solitaire problem is a satisfaction problem where a standard deck of playing cards — minus the ace of spades — is dealt into 17 piles of three cards. These cards are played one-by-one into a discard pile, which begins with only the ace of spades. Each card played must be one higher or one lower than the current top of the discard pile, and only the top card of each pile is playable. The task is to find a sequence of plays that ends with all cards on the discard pile. A model for this problem is:

```
% Card at position
array[1..52] of var 1..52: cardAt;
% Position of card
array[1..52] of var 1..52: position;
% Constraint A: Ace of Spades played first
constraint cardAt[1] == 1;
% Constraint B: Consecutive cards match
constraint forall(i in 1..51)
  ( table([cardAt[i], cardAt[i+1]],
          neighbours) );
% Constraint C: Link card-at and position
constraint inverse(cardAt, position);
% Constraint D: cards on top played first
constraint forall(i in 1..17, j in 1..2)
  ( position[layout[i,j]] <
    position[layout[i,j+1]] );
```

and is included in the MiniZinc distribution with (a) one variable for each card representing the position in the sequence it is played, and (b) one variable for each position in the sequence representing the card played at that position. These variables are linked by an `inverse` constraint (constraint C in the model). The other two significant constraints state that cards played in succession must match, i.e., their face values differ by exactly one (B), and that each card must be played before any card underneath it in its pile (D).

The model has an almost symmetry σ: cards with the same face value are symmetric, except for constraint D which states that for each pile `i`, the `j`th card is played before the one underneath (the `j+1`th). Relaxing the problem by removing D leads to the automatic detection of σ, that is, finding 37 pairs of cards that can be swapped and thus used as dominance mappings. The $scond$ for the dominance breaking constraint associated to σ, is simply σ(D). Since this is a satisfaction problem, the $ocond$ imposes an arbitrary ordering on the variables to break σ. This leads to the following dominance breaking constraint:

```
constraint
  (forall(i in 1..17, j in 1..2)
        (position_prime[layout[i,j]] <
         position_prime[layout[i,j+1]]))
  -> lex_lesseq(position,position_prime);
```

This constraint has two problems. First, it requires the solver to support a reified version of the lexicographical ordering constraint. We automatically fix this by observing that the `position` variables are constrained to be all different (by the `inverse` constraint) and, therefore, the lexicographical constraint can be reduced to `position[a] < position[b]`, where `a` and `b` are the two cards being swapped by the mapping. Second, the left hand side of the implication is too strong, causing very weak propagation. A single mapping affects at most two of the piles; the other piles are unchanged and so that part of the condition is known to be true and could be eliminated from the `forall`. We repair this automatically by omitting from the forall any part which is left syntactically identical by the symmetry. Additionally, after canonicalization the dominance breaking constraints do not refer to any variables introduced by the duplication process. Therefore, all of the duplicated variables and constraints are omitted. The resulting dominance constraints are similar to the ones manually identified by Chu and Stuckey[2012] for this problem.

Table 5 shows the effect on search reduction and running time for many instances of the model (again taken from [Chu and Stuckey, 2012]). Clearly, the dominance breaking constraints cause a large reduction in search space and running time. The automatic simplification of the dominance breaking constraint as described above is crucial – without it, the dominance breaking constraint does not reduce the search space.

5 Conclusion

We have presented an approach to automatically generate some dominance constraints by (a) automatically finding symmetries and almost symmetries for the satisfaction core of the problem, and (b) automatically generating dominance breaking constraints for these symmetries. While we have

Instance (id)	Nodes		Time (s)	
	None	Dom	None	Dom
blackhole-1	893	512	0.0	0.0
blackhole-2	>47,407,775	8,465,614	>1800.0	380.5
blackhole-3	2,114	662	0.1	0.0
blackhole-4	>54,421,468	1,162,184	>1800.0	56.7
blackhole-5	15,516	6,869	0.4	0.2
blackhole-6	0	0	0.0	0.0
blackhole-7	4,938	761	0.1	0.0
blackhole-8	0	0	0.0	0.0
blackhole-9	346,973	8,221	13.7	0.4
blackhole-10	0	0	0.0	0.0
blackhole-11	>66,299,020	>42,571,844	>1800.0	>1800.0
blackhole-12	14,031	2,897	0.5	0.2
blackhole-13	6,714	624	0.1	0.0
blackhole-14	>75,766,660	11,054,241	>1800.0	230.8
blackhole-15	>69,476,701	>40,358,616	>1800.0	>1800.0
blackhole-16	>81,184,247	4,015,958	>1800.0	138.4
blackhole-17	0	0	0.0	0.0
blackhole-18	312,905	30,560	9.7	1.2
blackhole-19	>61,435,110	48,829,969	>1800.0	1475.8
blackhole-20	916,488	62,131	18.0	1.8

Table 5: Node counts for blackhole solitaire.

shown that these constraints can significantly reduce the search effort and solving time, adding them is not always beneficial. In some cases they can add overhead without benefit and even enlarge the search tree. The experiments demonstrate the importance of removing redundant variables and constraints. This is an avenue for future work: more redundant parts should be automatically eliminated, perhaps by improving the canonicalization of the constraints so that common subexpression elimination [Rendl, 2010] can merge them.

References

[Chu and Stuckey, 2012] Geoffrey Chu and Peter J. Stuckey. A generic method for identifying and exploiting dominance relations. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 6–22. Springer Berlin Heidelberg, 2012.

[Cohen *et al.*, 2006] David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.

[Getoor *et al.*, 1997] Lise Getoor, Greger Ottosson, Markus Fromherz, and Björn Carlson. Effective redundant constraints for online scheduling. In *In Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 302–307, 1997.

[Mancini and Cadoli, 2005] Toni Mancini and Marco Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In Zucker and Saitta [2005], pages 165–181.

[Martin, 2005] Roland Martin. The Challenge of Exploiting Weak Symmetries. In Brahim Hnich, Mats Carlsson, François Fages, and Francesca Rossi, editors, *Proceedings of the International Workshop on Constraint Solving and Constraint Logic Programming*, volume 3978 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2005.

[Mears *et al.*, 2008] Christopher Mears, Maria Garcia de la Banda, Mark Wallace, and Bart Demoen. A novel approach for detecting symmetries in CSP models. In Laurent Perron and Michael A. Trick, editors, *CPAIOR*, volume 5015 of *LNCS*, pages 158–172. Springer, 2008.

[Mears *et al.*, 2009] Christopher Mears, Maria Garcia de la Banda, and Mark Wallace. On implementing symmetry detection. *Constraints*, 14(4):443–477, 2009.

[Meseguer and Torras, 2001] P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1-2):133–163, 2001.

[Nethercote *et al.*, 2007] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Principles and Practice of Constraint Programming-CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[Prestwich and Beck, 2004] Steven Prestwich and J. Christopher Beck. Exploiting dominance in three symmetric problems. In *in: Fourth International Workshop on Symmetry and Constraint Satisfaction Problems*, pages 63–70, 2004.

[Proll and Smith, 1997] Les Proll and Barbara Smith. ILP and constraint programming approaches to a template design problem. *INFORMS Journal of Computing*, 10:10–265, 1997.

[Puget, 2005] Jean-François Puget. Automatic detection of variable and value symmetries. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 475–489. Springer, 2005.

[Ramani and Markov, 2004] Arathi Ramani and Igor L. Markov. Automatically exploiting symmetries in constraint programming. In Boi Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 3419 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2004.

[Rendl, 2010] Andrea Rendl. *Effective compilation of constraint models*. PhD thesis, The University of St Andrews, 2010.

[Roy and Pachet, 1998] Pierre Roy and François Pachet. Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *ECAI98 Workshop on Non-binary Constraints*, pages 27–33, 1998.

[Van Hentenryck *et al.*, 2005] Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren. Compositional derivation of symmetries for constraint satisfaction. In Zucker and Saitta [2005], pages 234–247.

[Zucker and Saitta, 2005] Jean-Daniel Zucker and Lorenza Saitta, editors. *Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005, Proceedings*, volume 3607 of *Lecture Notes in Computer Science*. Springer, 2005.