Noname manuscript No.
(will be inserted by the editor)

Dominance Breaking Constraints

Geoffrey Chu · Peter J. Stuckey

the date of receipt and acceptance should be inserted later

Abstract Many constraint problems exhibit *dominance relations* which can be exploited for dramatic reductions in search space. Dominance relations are a generalization of symmetry and conditional symmetry. However, unlike symmetry breaking which is relatively well studied, dominance breaking techniques are not very well understood and are not commonly applied. In this paper, we present formal definitions of dominance breaking, and a generic method for identifying and exploiting dominance relations via *dominance breaking constraints*. We also give a generic proof of the correctness and compatibility of symmetry breaking constraints, conditional symmetry breaking constraints and dominance breaking constraints.

1 Introduction

In a constraint satisfaction or optimization problem, dominance relations describe pairs of assignments where one is known to be at least as good as the other with respect to satisfiability or the objective function. When such dominance relations are known, we can often prune off many of the solutions without changing the satisfiability or the optimal value of the problem. Many constraint problems exhibit dominance relations which can be exploited for significant speedups (e.g., [19,33, 2,24,32,28,7,15]).

Dominance relations are a generalization of symmetry and conditional symmetry and offer similar or greater potential for reductions in search space. Unlike symmetries however, dominance relations are not widely exploited. Dominance relations can be hard to identify, and there are few standard methods for exploiting them. It is also often hard to prove that a particular method is correct, especially when multiple dominance relations are being exploited simultaneously. These issues have been overcome in the case of symmetry, which is why symmetry breaking

An earlier version of this paper was published in 2012 [5]

National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
`{gchu,pjs}@csse.unimelb.edu.au`

is now standard and widely used. Dominance relations have been successfully applied in a number of problems, but their treatment is often highly problem specific and yields little insight as to how they can be generalized. In this paper, we seek to advance the usage of dominance relations by making the following contributions:

– We describe a generic method for identifying and exploiting a large class of dominance relations using *dominance breaking constraints*.
– We show that our method naturally produces symmetry breaking and conditional symmetry breaking constraints as well (since they are simply special cases of dominance breaking).
– We give a generic theorem proving the correctness and compatibility of all symmetry breaking, conditional symmetry breaking and dominance breaking constraints defined by our method.

The layout of the paper is as follows. In Section 2, we give our definitions. In Section 3, we describe our method of identifying and exploiting dominance relations using dominance breaking constraints. In Section 4, we describe how our method can be extended to generate symmetry and conditional symmetry breaking constraints as well. In Section 5 we discuss how dominance breaking interacts with search. In Section 6, we discuss related work. In Section 7, we provide experimental results. In Section 8, we conclude and discuss future work.

2 Definitions

In this section, we present our notations and definitions.

2.1 Constraint Programming

To facilitate rigorous proofs in the later sections, we will give our own definitions of variables, domains, constraints, constraint problems and dominance relations. These are slightly different from the standard definitions but are equivalent to them in practice.

Let $\equiv$ denote syntactical identity, $\Rightarrow$ denote logical implication and $\Leftrightarrow$ denote logical equivalence. We define variables and constraints in a problem independent way. A variable v is a mathematical quantity capable of assuming any value from a set of values called the *default domain* of v. Each variable is typed, e.g., Boolean or Integer, and its type determines its default domain, e.g., $\{0, 1\}$ for Boolean variables and $\mathbb{Z}$ for Integer variables. 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 default domain. A constraint c over a set of variables V is defined by a set of valuations $solns(c) \subseteq \Theta_V$. Given a valuation θ over $V' \supset V$, we say θ satisfies c if the restriction of θ onto V is in $solns(c)$. Otherwise, we say that θ violates c. A *domain* D over variables V is a set of *unary constraints*, one for each variable in V. In an abuse of notation, if a symbol A refers to a set of constraints $\{c_1, \ldots, c_n\}$, we will often also use the symbol A to refer to the constraint $c_1 \wedge \ldots \wedge c_n$. This allows us to avoid repetitive use of conjunction symbols.

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 n-ary constraints. A valuation θ over V is a *solution* of P if it satisfies every constraint in D and C. The aim of a CSP is to find a solution or to prove that none exist. In a *Constraint*

Table 1 Definitions of standard global constraints. In an abuse of notation, we consider false as 0 and true as 1.

Name and arguments	definition
$at\_least([x_1,\ldots,x_n],v,y)$	$(\sum_{i=1}^{n} x_i = v) \geq y$
$at\_most([x_1,\ldots,x_n],v,y)$	$(\sum_{i=1}^{n} x_i = v) \leq y$
$among([x_1,\ldots,x_n],v,y)$	$(\sum_{i=1}^{n} x_i = v) = y$
$all\_diff([x_1,\ldots,x_n])$	$\forall 1 \leq i < j \leq n, x_i \neq x_j$
$inverse([x_1,\ldots,x_n],[y_1,\ldots,y_n])$	$\forall 1 \leq i,j \leq n, x_i = j \Leftrightarrow y_j = i$
$gcc([x_1,\ldots,x_n],[v_1,\ldots,v_m],[y_1,\ldots,y_m])$	$\forall 1 \leq k \leq m, (\sum_{i=1}^{n} x_i = v_k) = y_k$
$element([x_1,\ldots,x_n],z,y)$	$y = x_z$
$table([x_1,\ldots,x_n],[[v_{1,1},\ldots,v_{1,n}],\ldots,[v_{m,1},\ldots,v_{m,n}]])$	$\exists k, \forall 1 \leq i \leq n, x_i = v_{k,i}$

Optimization Problem (COP) $P \equiv (V,D,C,f)$, we also have an objective function f mapping Θ_V to an ordered set, e.g., the set of integers $\mathbb{Z}$ or the set of real numbers $\mathbb{R}$, and we wish to minimize or maximize f over the solutions of P. In this paper, we deal with finite domain problems only, i.e., where the initial domain D constrains each variable to take values from a finite set of values.

Propagation based solvers (see e.g [37]) solve CSPs and COPs by interleaving tree search with inference. A *propagator* p_c for constraint c is a function mapping domains to domains s.t. $c \wedge D \Leftrightarrow c \wedge p_c(D)$, i.e., it performs inference based on c and the current domain D, pruning away any variable/value pairs that cannot be taken given c and D. Solving a CSP begins with the original problem at the root of the search tree. At each node in the search tree, we execute all the propagators until the domain reaches a fixed point. If some variable's domain becomes empty, then the subproblem has no solution and the solver backtracks. If all the variables are assigned and no constraint is violated, then a solution has been found and the solver can terminate. If inference is unable to detect either of the above two cases, then the solver divides the problem into a number of more constrained subproblems and searches each of those in turn. Typically this involves picking a variable x and branching on its values, e.g., trying $x = v$ for some value v in one child node and $x \neq v$ in the other child node, or trying $x \leq v$ in one child node and $x > v$ in the other.

CP solvers solve a minimization COP (V,D,C,f) by simply solving the CSP (V,D,C) to find a solution θ, and then solving the CSP $(V,D,C \wedge f < \theta(f))$ to find a new (better) solution θ. They repeat this until the final CSP has no solution in which case the last solution found is optimal.

One of the principal advantages of constraint programming is the ability to model directly with complex global constraints, that implement efficient propagators. Table 1 gives the definitions of a number of global constraints which we will use in this paper.

Later on we shall be interested in properties of constraints in order to simplify their handling. A property of interest is *monotonicity*

Definition 1 We say that an argument x_i in a constraint $c(x_1,\ldots,x_n)$ is *monotonically increasing* (resp. decreasing) w.r.t. c iff increasing (resp. decreasing) its value while keeping the other arguments fixed never cause c to go from satisfied to unsatisfied.

Example 1 The constraint $3x_1 + 4x_2 - 5x_3 \leq 8$ is monotonically increasing in x_3 and monotonically decreasing in x_1 and x_2. Similarly $at\_most([x_1,\ldots,x_n],v,y)$ is monotonically increasing in y, but neither monotonically increasing nor decreasing on any of the x_i. $\square$

We shall be particularly interested later in the paper on COPs with *lexicographic objective functions*. To fit this transparently into the usual COP definition we assume a function $lex : \mathbb{Z}^n \rightarrow \mathbb{Z}$ such that $lex(v_1, \ldots, v_n) < lex(v'_1, \ldots, v'_n) \Leftrightarrow (v_1 < v'_1) \vee (v_1 = v'_1 \wedge v_2 < v'_2) \vee \ldots \vee (\wedge_{i=1}^{n-1} v_i = v'_i \wedge v_n < v'_n)$.

2.2 Dominance

We define dominance relations over full valuations. We assume that all objective functions are to be minimized, and consider constraint satisfaction problems as constraint optimization problems with $f(\theta) = 0$ for all valuations θ.

Definition 2 A *dominance relation* $\prec$ for COP $P \equiv (V, D, C, f)$ is a *transitive* and *irreflexive* binary relation on Θ_V such that if $\theta_1 \prec \theta_2$, then either: 1) θ_1 is a solution of P and θ_2 is a not a solution of P, or 2) both θ_1 and θ_2 are solutions of P and $f(\theta_1) \leq f(\theta_2)$, or 3) both θ_1 and θ_2 are not solutions of P and $f(\theta_1) \leq f(\theta_2)$.

If $\theta_1 \prec \theta_2$, we say that θ_1 *dominates* θ_2. Note that we require our dominance relations to be irreflexive. This means that no loops can exist in the dominance relation, and makes it much easier to ensure the correctness of the method. The following theorem states that it is correct to prune all dominated assignments.

Theorem 1 *Given a finite domain COP $P \equiv (V, D, C, f)$, and a dominance relation $\prec$ for P, we can prune all assignments θ such that $\exists \theta'$ s.t. $\theta' \prec \theta$, without changing the satisfiability or optimal value of P.*

Proof Let θ_0 be an optimal solution. If θ_0 is pruned, then there exists some solution θ_1 s.t. $\theta_1 \prec \theta_0$. Then θ_1 must be a solution with $f(\theta_1) \leq f(\theta_0)$, so θ_1 is also an optimal solution. In general, if θ_i is pruned, then there must exist some θ_{i+1} s.t. $\theta_{i+1} \prec \theta_i$ and θ_{i+1} is also an optimal solution. Since $\prec$ is transitive and irreflexive, it is impossible for the sequence $\theta_0, \theta_1, \ldots$ to repeat. Then since there are finitely many solutions, the sequence must terminate in some θ_k which is an optimal solution and which is not pruned. $\square$

Theorem 1 relates to finite domain COP's. However, the proof for Theorem 1 generalizes trivially to the case of infinite domain COP's with finitely many solutions as well. On the other hand, it does not hold for infinite domain COP's with infinitely many solutions. E.g., consider $P \equiv (\{x\}, \{x \in \mathbb{Z}\}, \emptyset, x)$. We could define $\prec$ such that $\forall \theta_1, \theta_2$, $\theta_1 \prec \theta_2$ iff $\theta_1(x) < \theta_2(x)$. Since every possible valuation is dominated by something, pruning all dominated valuations ends up pruning all possible solutions. In this paper, we consider only finite domain problems, so Theorem 1 holds for the problems we consider.

Several previous definitions of dominance (e.g., [10,22]) relate search nodes rather than valuations. We can extend a precedence relation $\prec$ over valuations to relate search nodes in the obvious way.

Definition 3 Let D_1 and D_2 be the domains from two different search nodes. If $\forall \theta_2 \in solns(D_2), \exists \theta_1 \in solns(D_1)$ s.t. $\theta_1 \prec \theta_2$, then we define $D_1 \prec D_2$.

Clearly if $D_1 \prec D_2$, Theorem 1 tells us that we can safely prune the search node with D_2. We call the pruning allowed by Theorem 1 *dominance breaking* in keeping with *symmetry breaking* for symmetries.

Example 2 Consider a simple problem with domain $x_i \in \{1, \ldots, 10\}$, constraint *all_diff*$([x_1, \ldots, x_{10}])$ and objective function $\sum_{i=1}^{10} i{*}x_i$ to be minimized. The search node n given by the decisions $x_1 = 2, x_2 = 1$ dominates the search node n' given by the decisions $x_1 = 1, x_2 = 2$, as no matter how we label the remaining variables in n', the corresponding assignment in n with the same values for $x_3, \ldots, x_{10}$ will always have a better objective value since $1 * 2 + 2 * 1 < 1 * 1 + 2 * 2$.

Dominance relations can be derived either statically before search or dynamically during search in order to prune the search space. It is easy to see that static symmetry breaking (e.g., [9,13]) is a special case of static dominance breaking. For example, consider the lex-leader method of symmetry breaking. Suppose S is a symmetry group of problem P. Suppose $l(\theta)$ is the lexicographical function being used in the lex-leader method. We can define a dominance relation: $\forall \sigma \in S, \forall \theta, \sigma(\theta) \prec \theta$ if $l(\sigma(\theta)) < l(\theta)$. Then applying Theorem 1 to $\prec$ gives the lex-leader symmetry breaking constraint (i.e., prune all solutions which are not the lex-leader in their equivalence class). Similarly, dynamic symmetry breaking techniques such as Symmetry Breaking During Search [18] and Symmetry Breaking by Dominance Detection [10,14] are special cases of dynamic dominance breaking. Nogood learning techniques such as Lazy Clause Generation [30,11] and Automatic Caching via Constraint Projection [6] are also examples of dynamic dominance breaking. We will discuss these two methods in more detail in Section 6.

Just as in the case of symmetry breaking (see e.g. [12]), it is generally *incorrect* to simultaneously post dominance breaking constraints for multiple dominance relations. This is because dominance relations only ensure that one assignment is at least as good as the other (not strictly better than), thus when we have multiple dominance relations, we could have loops such as $\theta_1 \prec_1 \theta_2$ and $\theta_2 \prec_2 \theta_1$, and posting the dominance breaking constraint for both $\prec_1$ and $\prec_2$ would be wrong. We have to take care when breaking symmetries, conditional symmetries and dominances that all the pruning we perform are compatible with each other. As we shall show below, one of the advantages of our method is that all the symmetry breaking, conditional symmetry breaking and dominance breaking constraints defined by our method are provably compatible.

Dominance breaking constraints can be particularly useful in optimization problems, because they provide a completely different and complementary kind of pruning to the branch and bound paradigm. In the branch and bound paradigm, the only way to show that a partial assignment is suboptimal is to prove a sufficiently strong bound on its objective value. Proving such bounds can be highly expensive, especially if the model does not propagate strong bounds on the objective. In the worst case, further search is required, which can take an exponential amount of time. On the other hand, dominance breaking can prune a partial assignment without having to prove any bounds on its objective value at all, since it only needs to know that the partial assignment is suboptimal. Once dominance relations expressing conditions for suboptimality are found and proved, the only cost in the search is to check whether a partial assignment is dominated, which can often be much lower than the cost required to prove a sufficiently strong bound to prune the partial assignment.

3 Identifying and Exploiting Dominance Relations

3.1 Overview of method

We now describe a generic method for identifying and exploiting a fairly large class of dominance relations using dominance breaking constraints. The idea is to use mappings σ from valuations to valuations to construct dominance relations. Given a mapping σ, we ask: under what conditions does σ map a solution to a better solution? If we can find these conditions, then we can build a dominance relation using these conditions and exploit it by posting a dominance breaking constraint. More formally:

Step 1 Choose a set S of mappings $\sigma : \Theta_V \to \Theta_V$ which are likely to map solutions to better solutions.

Step 2 For each $\sigma \in S$, find a constraint $\mathit{scond}(\sigma)$ s.t. if $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{scond}(\sigma))$, then $\sigma(\theta) \in \mathit{solns}(C \wedge D)$.

Step 3 For each $\sigma \in S$, find a constraint $\mathit{ocond}(\sigma)$ s.t. if $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{ocond}(\sigma))$, then $f(\sigma(\theta)) < f(\theta)$.

Step 4 For each $\sigma \in S$, post the dominance breaking constraint $\mathit{db}(\sigma) \equiv \neg(\mathit{scond}(\sigma) \wedge \mathit{ocond}(\sigma))$.

The constraints $\mathit{scond}(\sigma)$ ensure that σ maps solutions to solutions, while $\mathit{ocond}(\sigma)$ ensures that σ maps valuations to better valuations. Finally $\mathit{db}(\sigma)$ ensures that solutions that are dominated are eliminated.

The following theorem proves the correctness of this method.

Theorem 2 *Given a finite domain COP $P \equiv (V, D, C, f)$, a set of mappings S, and for each mapping $\sigma \in S$ constraints $\mathit{scond}(\sigma)$ and $\mathit{ocond}(\sigma)$ satisfying: $\forall \sigma \in S$, if $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{scond}(\sigma))$, then $\sigma(\theta) \in \mathit{solns}(C \wedge D)$, and: $\forall \sigma \in S$, if $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{ocond}(\sigma))$, then $f(\sigma(\theta)) < f(\theta)$, we can add all of the dominance breaking constraints $\mathit{db}(\sigma) \equiv \neg(\mathit{scond}(\sigma) \wedge \mathit{ocond}(\sigma))$ to P without changing its satisfiability or optimal value.*

Proof Construct a binary relation $\prec$ as follows. For each σ, for each $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{scond}(\sigma) \wedge \mathit{ocond}(\sigma))$, define $\sigma(\theta) \prec \theta$. Now, take the transitive closure of $\prec$. We claim that $\prec$ is a dominance relation. It is transitive by construction. Also, by construction, $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{scond}(\sigma) \wedge \mathit{ocond}(\sigma))$ guarantees that $\sigma(\theta)$ is a solution and that $f(\sigma(\theta)) < f(\theta)$. Thus $\forall \theta_1, \theta_2, \theta_1 \prec \theta_2$ implies that θ_1 and θ_2 are solutions, and that $f(\theta_1) < f(\theta_2)$. This means that $\prec$ is irreflexive and satisfies all the properties of a dominance relation, thus by Theorem 1, we can prune any $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{scond}(\sigma) \wedge \mathit{ocond}(\sigma))$ for any σ without changing the satisfiability or optimality of P. Thus it is correct to add $\mathit{db}(\sigma)$ for any σ to P. $\square$

Note that there are no restrictions on σ. It does not have to be injective or surjective. The $\mathit{db}(\sigma)$ are guaranteed to be compatible because they all obey the same strict ordering imposed by the objective function f, i.e., they prune a solution only if a solution with strictly better f value exists. Of course not every $\mathit{db}(\sigma)$ will be valuable to add to the problem, $\mathit{db}(\sigma)$ may be equivalent to *true* or a complex constraint implemented by reification which barely prunes.

We illustrate the method with two simple examples before we go into more details.

Example 3 Consider the Photo problem. A group of people wants to take a group photo where they stand in one line. Each person has preferences regarding who they want to stand next to. We want to find the arrangement which satisfies the most preferences.

We can model this as follows. Let $x_i \in \{1, \ldots, n\}$ for $i = 1, \ldots, n$ be variables where x_i represents the person in the ith place. Let p be a 2d integer array where $p[i][j] = p[j][i] = 2$ if person i and j both want to stand next to each other, $p[i][j] = p[j][i] = 1$ if only one of them wants to stand next to the other, and $p[i][j] = p[j][i] = 0$ if neither want to stand next to each other. The only constraint is: $\mathit{all\_diff}([x_1, \ldots, x_n])$. The objective function to be minimized is given by: $f = -\sum_{i=1}^{n-1} p[x_i][x_{i+1}]$.

Step 1 Since this is a sequence type problem, mappings which permute the sequence in some way are likely to map solutions to solutions. For simplicity, consider the set of mappings which flip a subsequence of the sequence, i.e., $\forall i < j, \sigma_{i,j}$ maps x_i to x_j, x_{i+1} to x_{j-1}, …, x_j to x_i and each other variable to itself.

Step 2 We want to find the conditions under which σ maps solutions to solutions. Since all of these σ are symmetries of $C \wedge D$, we do not need any conditions and it is sufficient to set $\mathit{scond}(\sigma_{i,j}) \equiv \mathit{true}$.

Step 3 We want to find the conditions under which $f(\sigma_{i,j}(\theta)) < f(\theta)$. If we compare the LHS and RHS, it is clear that the only difference is the terms $p[x_{i-1}][x_j]$, $p[x_i][x_{j+1}]$ on the LHS and the terms $p[x_{i-1}][x_i]$, $p[x_j][x_{j+1}]$ on the RHS. So it is sufficient to set $\mathit{ocond}(\sigma_{i,j}) \equiv p[x_{i-1}][x_j] + p[x_i][x_{j+1}] > p[x_{i-1}][x_i] + p[x_j][x_{j+1}]$.

Step 4 For each $\sigma_{i,j}$, we can post the dominance breaking constraint:

$$\neg(p[x_{i-1}][x_j] + p[x_i][x_{j+1}] > p[x_{i-1}][x_i] + p[x_j][x_{j+1}]).$$

These dominance breaking constraints ensure that if some subsequence of the assignment can be flipped to improve the objective, then the assignment is pruned. □

Example 4 Consider the 0-1 knapsack problem maximizing the value of items chosen from a set S within a weight limit W. Then $x_i, i \in S$ are 0-1 variables, and we have constraint $\sum_{i \in S} w_i x_i \leq W$ and we have objective $f = -\sum_{i \in S} v_i x_i$, where w_i is the (constant) weight of item i and v_i is the (constant) price of item i.

Step 1 Consider mappings which swap the values of two variables, i.e., $\forall i < j, \sigma_{i,j}$ swaps x_i and x_j.

Step 2 A sufficient condition for $\sigma_{i,j}$ to map the current solution to another solution is: $\mathit{scond}(\sigma_{i,j}) \equiv w_i x_j + w_j x_i \leq w_i x_i + w_j x_j$. Rearranging, we get: $(w_i - w_j)(x_i - x_j) \geq 0$.

Step 3 A sufficient condition for $\sigma_{i,j}$ to map the current solution to an assignment with a better objective value is: $\mathit{ocond}(\sigma_{i,j}) \equiv v_i x_j + v_j x_i > v_i x_i + v_j x_j$. Rearranging, we get: $(v_i - v_j)(x_i - x_j) < 0$.

Table 2 Standard mappings we can try with the method.

Mapping	effect
$\mathit{var\_perm}_\pi$, π a permutation	maps x_i to $x_{\pi(i)}$
$\mathit{var\_swap}_{i,j}$	swaps x_i and x_j
$\mathit{row\_swap}_{i,j}$	swaps $x_{i,k}$ with $x_{j,k}$ for all k
$\mathit{col\_swap}_{i,j}$	swaps $x_{k,i}$ with $x_{k,j}$ for all k
$\mathit{shift\_sub\_seq}_{i,j,k}$ where $i < k$	maps x_m to x_{m+k-i} for $i \leq m \leq j$, maps x_m to $x_{m-j-1+i}$ for $j+1 \leq m \leq k+j-i$
$\mathit{flip\_sub\_seq}_{i,j}$	maps x_k to x_{j+i-k} for $i \leq k \leq j$
$\mathit{val\_perm}_{V',\pi}$, $V' \subseteq V$, π a permutation	maps the value v to $\pi(v)$ on all vars in V'
$\mathit{shift\_val\_}V', i$	increase the value of all $x_k \in V'$ by i

Step 4 For each $\sigma_{i,j}$, we can post the dominance breaking constraint: $db(\sigma_{i,j}) \equiv \neg(\mathit{scond}(\sigma_{i,j}) \wedge \mathit{ocond}(\sigma_{i,j}))$. After simplifying, we have $db(\sigma_{i,j}) \Leftrightarrow x_i \leq x_j$ if $w_i \geq w_j$ and $v_i < v_j$, $db(\sigma_{i,j}) \Leftrightarrow x_i \geq x_j$ if $w_i \leq w_j$ and $v_i > v_j$, and $db(\sigma_{i,j}) \Leftrightarrow \mathit{true}$ for all other cases.

These dominance breaking constraints ensure that if one item has worse price and greater or equal weight to another, then it cannot be chosen without choosing the other also. $\square$

3.2 Step 1: Finding Appropriate Mappings σ

In general, we want to find σ's such that $\mathit{scond}(\sigma)$ and $\mathit{ocond}(\sigma)$ are as small and simple as possible, as this will lead to dominance breaking constraints that are easier to propagate and prune more. So we want σ such that it often maps a solution to a better solution. Mappings σ which are symmetries or almost symmetries of the problem make good candidates, since their $\mathit{scond}(\sigma)$ will be simple, and all else being equal, there is a 50% chance that it will map the solution to one with a better objective value.

We can try all the common candidates for symmetries. For example, if the variables or the values represent the same type of thing, we can try swapping them. E.g., if x_i represents truck i's load, we could try swapping these variables. Or if $x_i \in \{1, \ldots, 10\}$ where each value represents a different room, we could try swapping the values (rooms). Many problems are representable as 2 dimensional matrices (see for example [12]) where each row represents one thing and each column represent another. In such problems, we can try swapping two rows or two columns in the matrix. Another common type of combinatorial problem is a sequencing problem where we are trying to find an order of a certain set of things which satisfies some constraints or optimizes some objective. In such sequence type problems, we can try flipping or moving a subsequence.

Mappings which are likely to map an assignment to one with better objective value are also good candidates, since their $\mathit{ocond}(\sigma)$ will be simple. For example, in scheduling problems minimizing makespan, we can try shifting items forward in the schedule. There may also be problem specific σ's that we can try. Table 2 shows a list of standard mappings we can try.

Table 3 σ and c such that σ maps c to itself.

σ	c
var_perm_π	$y = (\geq, \leq) f(x_1, \ldots, x_n)$, where $\pi(y) = y$, $\forall i, \exists j, \pi(x_i) = x_j$, f is a symmetric function, e.g., min, max, and, or, sum
$val\_perm_{V',\pi}$	$x_i = (\neq) x_j$ where $x_i, x_j \in V'$
$val\_perm_{V',\pi}$	$among([x_1, \ldots, x_n], v, y)$, where $x_i, v \in V'$, $y \notin V'$
$val\_perm_{V',\pi}$	$element([x_1, \ldots, x_n], z, y)$, where $y, x_i \in V'$, $z \notin V'$
$val\_perm_{V',\pi}$	$table([x_1, \ldots, x_n], [[z_{1,1}, \ldots, z_{1,n}], \ldots, [z_{m,1}, \ldots, z_{m,n}]])$, where $x_i, z_{i,j} \in V'$

3.3 Step 2: Finding $scond(\sigma)$

First, we need to define how constraints are mapped under arbitrary mappings from valuations to valuations.

Definition 4 Given a mapping $\sigma : \Theta_V \to \Theta_V$, we can extend σ to map constraints to constraints as follows. Given a constraint c, $\sigma(c)$ is defined as a constraint over V such that θ satisfies $\sigma(c)$ iff $\sigma(\theta)$ satisfies c.

While it is easy to define $\sigma(c)$, $\sigma(c)$ may or may not be a simple logical expression.

Example 5 We illustrate a number of constraints $\sigma(c)$ that result from applying a mapping σ to constraint c:

– Suppose $c \equiv x_1 + 2x_2 + 3x_3 \geq 10$, and σ swaps x_1 and x_3, then $\sigma(c) \equiv x_3 + 2x_2 + 3x_1 \geq 10$.
– Suppose $c \equiv (x_1, x_2) \in \{(1, 1), (2, 3), (3, 1)\}$, and σ permutes the values (1, 2, 3) to (2, 3, 1) on x_1 and x_2, then $\sigma(c) \equiv (x_1, x_2) \in \{(3, 3), (1, 2), (2, 3)\}$.
– Suppose $c \equiv x_1 + 2x_2 \geq 5$ and σ swaps the values 1 and 2, then $\sigma(c) \equiv (x_1 = 1 \wedge x_2 = 1) \vee (x_1 = 2 \wedge x_2 = 1) \vee (x_1 \neq 1 \wedge x_1 \neq 2 \wedge x_2 \neq 1 \wedge x_2 \neq 2 \wedge x_1 + 2x_2 \geq 5)$ which does not simplify at all.

$\square$

In general, σ can either map c to itself, map it to another easily expressible constraint, or map it to something very complicated. Table 3 shows σ and c's where σ maps c to itself. Table 4 shows σ and c's where σ maps c to something different but still easily expressible. Any constraint built up from the above primitive constraints maps in the obvious way, i.e., $\sigma(c_1 \wedge \ldots \wedge c_n) \Leftrightarrow \sigma(c_1) \wedge \ldots \wedge \sigma(c_n)$. So for example, we can also map *at_least*, *at_most*, *all_diff* and *gcc* (which are built from *among*), or the *regular* constraint [31] (which can be built from *table*), under arbitrary value permutations. Many constraints do not map to nice expressions under value permutations, e.g., linear constraints, multiplication constraints or division constraints. For such constraints, $\sigma(c)$ cannot easily be expressed or propagated.

Example 6 The constraint $c \equiv at\_least([x_1, \ldots, x_n], v, y)$ can be defined as $among([x_1, \ldots, x_n], v, z) \wedge z \geq y$. Hence if σ is $val\_perm_{V',\pi}$ where $x_i \in V'$ and $y \notin V'$ we can define $\sigma(c)$ as $among([x_1, \ldots, x_n], \pi(v), z) \wedge z \geq y$. $\square$

We can now calculate $scond(\sigma)$ with the help of the above definition. Note that while $\sigma(c)$ is uniquely defined, $scond(\sigma)$ is only a *sufficient* condition for something to hold, thus there is plenty of leeway for us to pick between different options. We

Table 4 $\sigma(c)$ for various σ and c.

σ	c	$\sigma(c)$
var_perm_{π}	any constraint $c(x_{i_1}, \dots, x_{i_n})$	$c(x_{\pi(i_1)}, \dots, x_{\pi(i_n)})$
$val\_perm_{V',\pi}$	$x_i = (\neq) v$ where $x_i \in V'$	$x_i = (\neq) \pi(v)$
$val\_perm_{V',\pi}$	$among([x_1, \dots, x_n], v, y)$, where $x_i \in V'$, $y \notin V'$	$among([x_1, \dots, x_n], \pi(v), y)$
$val\_perm_{V',\pi}$	$element([v_1, \dots, v_m], z, y)$, where $y \in V', z \notin V'$	$element([\pi(v_1), \dots, \pi(v_n)], z, y)$
$val\_perm_{V',\pi}$	$table([x_1, \dots, x_n], [[v_{1,1}, \dots, v_{1,n}], \dots,$ $[v_{m,1}, \dots, v_{m,n}]])$	$table([x_1, \dots, x_n], [[\pi(v_{1,1}), \dots, \pi(v_{1,n})],$ $\dots, [\pi(v_{m,1}), \dots, \pi(v_{m,n})]])$

require $scond(\sigma)$ to be a constraint such that $scond(\sigma) \wedge C \wedge D \Rightarrow \sigma(C \wedge D)$. We can construct $scond(\sigma)$ in a piecewise fashion by considering each constraint in the problem in turn. Let $scond(\sigma, c)$ be a constraint such that $C \wedge D \wedge scond(\sigma, c) \Rightarrow \sigma(c)$. Then we can construct $scond(\sigma)$ as $\wedge_{c \in C \wedge D} scond(\sigma, c)$. Naively, we can set $scond(\sigma, c) = \sigma(c)$. This is clearly correct. However, we can make use of the existing constraints $C \wedge D$ to simplify $scond(\sigma, c)$ in order to get a stronger (more general) or simpler (faster to propagate) condition. One special case is subsumption. For each $c \in C \wedge D$, if $\sigma(c)$ is already in $C \wedge D$, then we can simply set $scond(\sigma, c) = true$, since $C \wedge D$ already unconditionally implies $\sigma(c)$.

Example 7 Suppose $C \equiv \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1, x_1 \neq x_4\}$, and σ swaps x_1 and x_2. $\sigma(x_1 \neq x_2) = x_1 \neq x_2$, which is already in C. $\sigma(x_2 \neq x_3) = x_1 \neq x_3$ which is already in C. $\sigma(x_3 \neq x_1) = x_3 \neq x_2$ which is already in C. $\sigma(x_1 \neq x_4) = x_2 \neq x_4$ which is not in C. So we can set $scond(\sigma) = true \wedge true \wedge true \wedge x_2 \neq x_4$. $\square$

This is the reason why σ's which are symmetries or almost symmetries of the problem make good candidates for the method, as most or all of the $\sigma(c)$ are already in $C \wedge D$ and thus we will have a simple $scond(\sigma)$. If $\sigma(c)$ is not subsumed by $C \wedge D$, we can still potentially simplify it using $C \wedge D$. The simplest case here is to use c itself to simplify $scond(\sigma, c)$, i.e., we want to find $scond(\sigma, c)$ such that $c \wedge scond(\sigma, c) \Rightarrow \sigma(c)$. There is a potential trade off between strength and speed. Simpler $scond(\sigma)$ could be propagated faster, but may be weaker in terms of what can ultimately be pruned by the dominance breaking constraint.

Example 8 Consider $c \equiv x_1 + x_2 + x_3 \leq 3$. Suppose σ swaps x_3 with x_4. One possible $scond(\sigma, c)$ is simply $\sigma(c)$ which is $x_1 + x_2 + x_4 \leq 3$. Alternatively, another sufficient condition is $x_4 \leq x_3$, since $x_1 + x_2 + x_3 \leq 3 \wedge x_4 \leq x_3 \Rightarrow x_1 + x_2 + x_4 \leq 3$. The first condition is stronger (more general), but may cost more to propagate. The second condition is weaker (less general), but may be faster to propagate. $\square$

Simplifications are important from a practical point of view, as we may be posting many dominance breaking constraints, and having many extra reified propagators to propagate $\sigma(c)$ may be far too costly. Table 5 gives some possible simplifications. This table is by no means exhaustive.

Note that depending on the values of the constants in the problem, the expression shown in the table will simplify even further. For example, on the sixth line, if $a_1 - a_2 > 0$, then it simplifies to $x_1 \geq x_2$.

3.4 Step 3: Finding $ocond(\sigma)$

We assume that the objective function $f(\theta)$ is defined over all assignments (not just solutions). We first give a few definitions.

Table 5 Possible $scond(\sigma, c)$ for various σ and c.

σ	c	$scond(\sigma, c)$
swap x_1 and x_k	$c(x_1, \ldots, x_n)$, c an arbitrary constraint	$x_1 = x_k$
swap x_1 and x_{n+1}	$c(x_1, \ldots, x_n)$, x_1 monotonically increasing in c	$x_1 \leq x_{n+1}$
swap x_1 and x_{n+1}	$c(x_1, \ldots, x_n)$, x_1 monotonically decreasing in c	$x_1 \geq x_{n+1}$
swap x_1 and x_2	$c(x_1, \ldots, x_n)$, x_1 mono inc, x_2 mono dec in c	$x_1 \leq x_2$
swap y_1 and y_2	$y_1 = f(x_1, \ldots, x_n)$, f a function	$y_1 = y_2$
swap x_1 and x_2	$\sum_{i=1}^{n} a_i x_i \leq k$	$(a_1 - a_2)(x_1 - x_2) \geq 0$
swap x_1 and x_{n+1}	$element([x_1, \ldots, x_n], i, y)$	$i = 1 \rightarrow x_1 = x_{n+1}$
swap x_1 and x_2	$element([x_1, \ldots, x_n], i, y)$	$(i = 1 \lor i = 2) \rightarrow x_1 = x_2$

Definition 5 Given a function σ mapping assignments to assignments, we extend σ to map functions over assignments to functions over assignments as follows: $\forall \theta,\ \sigma(f)(\theta) = f(\sigma(\theta))$.

Definition 6 Given two functions mapping assignments to the reals f and g, we use $f < g$ to denote a constraint such that: θ satisfies $f < g$ iff $f(\theta) < g(\theta)$.

For each σ, we want to find a sufficient condition $ocond(\sigma)$ so that if a solution satisfied $ocond(\sigma)$, then σ maps it to an assignment with a strictly better objective value. A necessary and sufficient condition is: $C \land D \land ocond(\sigma) \Rightarrow \sigma(f) < f$. For sum type objective functions, we can typically just set $ocond(\sigma) \equiv \sigma(f) < f$. For example, in both the Photo and Knapsack examples above, we simplified the constraint $\sigma(f) < f$. Depending on the mapping, many terms may be unchanged and can be eliminated, leading to a relatively simple $ocond(\sigma)$. For max/min type objective functions, besides finding an exact condition, we can also find a sufficient condition $ocond(\sigma)$ by piecewise comparison of each term in the max/min expression. For example, if $f \equiv min(x_1, \ldots, x_n)$, and $f' \equiv min(x'_1, \ldots, x'_n)$ we could set $ocond(\sigma) \equiv \land(x'_i < x_i)$, or, given some mapping π from $[1..n]$ to itself which depended on σ, we could set $ocond(\sigma) \equiv \land(x_{\pi(i)} < x_i)$.

3.5 Step 4: Posting the Dominance Breaking Constraint

Once we have found $scond(\sigma)$ and $ocond(\sigma)$, we can construct the dominance breaking constraint $db(\sigma) \equiv \neg(scond(\sigma) \land ocond(\sigma))$ and simplify it as much as possible. If it is simple enough to implement efficiently, we can add it to the problem. If not, we can either weaken it into a simpler form, or simply ignore it, as it is not required for the correctness of the method. It is quite common that the dominance breaking constraint for different σ's will have common subexpressions. Taking advantage of common subexpressions improves propagation for CP [36], and we can use this to make the implementation of the dominance breaking constraints more efficient. For example, in the dominance breaking constraints for the Photo problem given in Example 3, the expression $p[x_{i-1}][x_i]$ will appear in the dominance breaking constraint for multiple values of j, so common subexpression elimination will be able to replace these with a single intermediate variable.

4 Generating Symmetry and Conditional Symmetry Breaking Constraints

The method described so far only finds dominance breaking constraints which prune a solution when its objective value is *strictly* worse than another. We can do better than this, as there are often pairs of solutions which have equally good objective value and we may be able to prune many of them. Exploiting such sets of equally good pairs of solution is called symmetry breaking and conditional symmetry breaking. We show that with a slight alteration, our method will generate dominance breaking constraints that will also break symmetries and conditional symmetries.

We modify the method as follows. We add in a Step 0, and alter Step 3 slightly.

Step 0 Choose a refinement of the objective function f' with the property that $\forall \theta_1, \theta_2, f(\theta_1) < f(\theta_2)$ implies $f'(\theta_1) < f'(\theta_2)$.

*Step 3** For each σ, find a constraint $\mathit{ocond}(\sigma)$ s.t. if $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{ocond}(\sigma))$, then $f'(\sigma(\theta)) < f'(\theta)$.

The following theorem shows the correctness of the altered method.

Theorem 3 *Given a finite domain COP $P \equiv (V, D, C, f)$, a refinement of the objective function f' satisfying $\forall \theta_1, \theta_2, f(\theta_1) < f(\theta_2)$ implies $f'(\theta_1) < f'(\theta_2)$, a set of mappings S, and for each mapping $\sigma \in S$ constraints $\mathit{scond}(\sigma)$ and $\mathit{ocond}(\sigma)$ satisfying: $\forall \sigma \in S$, if $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{scond}(\sigma))$, then $\sigma(\theta) \in \mathit{solns}(C \wedge D)$, and: $\forall \sigma \in S$, if $\theta \in \mathit{solns}(C \wedge D \wedge \mathit{ocond}(\sigma))$, then $f'(\sigma(\theta)) < f'(\theta)$, we can add all of the dominance breaking constraints $\mathit{db}(\sigma) \equiv \neg(\mathit{scond}(\sigma) \wedge \mathit{ocond}(\sigma))$ to P without changing its satisfiability or optimal value.*

Proof The proof is almost identical to that of Theorem 2 by simply replacing f by f'. The critical difference arises in proving that $\prec$ is a dominance relation. Now $\forall \theta_1, \theta_2, \theta_1 \prec \theta_2$ implies that θ_1 and θ_2 are solutions, and that $f'(\theta_1) < f'(\theta_2)$ and hence also $f'(\theta_1) \leq f'(\theta_2)$. By the definition of f', $f(\theta_2) < f(\theta_1) \Rightarrow f'(\theta_2) < f'(\theta_1)$ and indeed also the contrapositive $f'(\theta_2) \geq f'(\theta_1) \Rightarrow f(\theta_2) \geq f(\theta_1)$, and thus $f'(\theta_1) \leq f'(\theta_2) \Rightarrow f(\theta_1) \leq f(\theta_2)$. Clearly then $f(\theta_1) \leq f(\theta_2)$. Since $f'(\theta_1) < f'(\theta_2)$ it follows that $\prec$ is irreflexive and since $f(\theta_1) \leq f(\theta_2)$ it satisfies all the properties of a dominance relation. □

The $\mathit{db}(\sigma)$ are guaranteed to be compatible because they all obey the same strict ordering imposed by the refined objective function f'. That is, they prune a solution only if a solution with strictly better f' value exists. Theorem 3 is useful because it is generally quite difficult to tell whether different symmetry, conditional symmetry or dominance breaking constraints are compatible. There are lots of examples in the literature where individual dominance breaking constraints are proved correct, but no rigorous proof is given that they are correct when used together (e.g., [15, 7, 17]). The symmetry, conditional symmetry or dominance breaking constraints defined by our method are guaranteed to be compatible by Theorem 3, thus the users of the method do not need to prove anything themselves.

The most common type of objective refinement is a lexicographical tie breaking using additional properties of the solutions. We set $f' = \mathit{lex}(f, p_1, \ldots, p_n)$,[1] where $p_i : \Theta_V \to \mathbb{Z}$ are some additional properties. f' orders the solutions first by their objective value, then tie breaks by the value of p_1, then tie breaks by the value of p_2, etc. Clearly, $f(\theta_1) < f(\theta_2)$ implies $f'(\theta_1) < f'(\theta_2)$ so it is a refinement. Recall that we want to set $\mathit{ocond}(\sigma) \equiv \sigma(f') < f'$. In general, we have $\sigma(\mathit{lex}(f, p_1 \ldots, p_n)) <$

[1] We write $f' = \mathit{lex}(f, p_1, \ldots, p_n)$ as shorthand for $f'(\theta) = \mathit{lex}(f(\theta), p_1(\theta), \ldots, p_n(\theta))$.

$\mathit{lex}(f, p_1, \ldots, p_n) \Leftrightarrow (\sigma(f) < f) \vee (\sigma(f) = f \wedge \sigma(p_1) < p_1) \vee \ldots \vee (\sigma(f) = \mathit{obj} \wedge \wedge_{i=1}^{n-1}\sigma(p_i) = p_i \wedge \sigma(p_n) < p_n)$. Thus $\mathit{ocond}(\sigma)$ will be the disjunction of a number of terms. The first of these $(\sigma(f) < f)$ will result in a term in the dominance breaking constraint expressing strict improvement in the objective. The remaining terms $(\sigma(f) = \mathit{obj} \wedge \wedge_{i=1}^{k-1}\sigma(p_i) = p_i \wedge \sigma(p_k) < p_k)$ will result in terms expressing (conditional) symmetry breaking.

Note that for many mappings σ and refined objectives $f' = \mathit{lex}(f, p_1, \ldots, p_n)$, the symmetry breaking part is well studied and there exist standard ways to model and propagate them. In such cases, we can just directly reuse the existing symmetry breaking constraints rather than manually recreating it. To be more precise, if we already have a standard lex-leader symmetry breaking constraint $\mathit{sb}(\sigma)$ implementing $\neg(\sigma(\mathit{lex}(p_1, \ldots, p_n)) < \mathit{lex}(p_1, \ldots, p_n))$, then we can set $\mathit{ocond}(\sigma) \equiv f' < f \vee (f' = f \wedge \neg(\mathit{sb}(\sigma))$. Then $\mathit{db}(\sigma) \equiv \neg(\mathit{scond}(\sigma) \wedge f' < f) \wedge (\mathit{scond}(\sigma) \wedge f' = f \rightarrow \mathit{sb}(\sigma))$. The first term is the strict dominance breaking constraint. The second term is a conditional symmetry breaking constraint making use of the standard symmetry breaking constraint. We now illustrate the altered method with some examples.

Example 9 Consider the Photo problem from Example 3. Suppose that in Step 0, instead of setting $f' = f$, we set $f' = \mathit{lex}(f, x_1, \ldots, x_n)$. Now, consider what happens in Step 3*. We have $\forall i < j$,

$$\begin{aligned}
&\mathit{ocond}(\sigma_{i,j}) \equiv \sigma_{i,j}(f') < f' \\
\Leftrightarrow\ &\sigma_{i,j}(\mathit{lex}(f, x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n)) < \mathit{lex}(f, x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) \\
\Leftrightarrow\ &\mathit{lex}(\sigma_{i,j}(f), x_1, \ldots, x_j, \ldots, x_i, \ldots, x_n) < \mathit{lex}(f, x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) \\
\Leftrightarrow\ &\mathit{lex}(\sigma_{i,j}(f), x_j, x_i) < \mathit{lex}(f, x_i, x_j) \\
\Leftrightarrow\ &\sigma_{i,j}(f) < f \vee (\sigma_{i,j}(f) = f \wedge x_j < x_i) \vee (\sigma_{i,j}(f) = f \wedge x_j = x_i \wedge x_i < x_j) \\
\tilde{\Leftrightarrow}\ &\sigma_{i,j}(f) < f \vee (\sigma_{i,j}(f) = f \wedge x_j < x_i) \\
\Leftrightarrow\ &(p[x_{i-1}][x_j] + p[x_i][x_{j+1}] > p[x_{i-1}][x_i] + p[x_j][x_{j+1}]) \vee \\
&(p[x_{i-1}][x_j] + p[x_i][x_{j+1}] = p[x_{i-1}][x_i] + p[x_j][x_{j+1}] \wedge x_j < x_i)
\end{aligned}$$

The first equivalence follows from definition of f'. The second holds by definition of $\sigma_{i,j}$. The third equivalence by the properties of *lex*. The fourth equivalence from the definition of *lex*. The fifth (pseudo-)equivalence holds since $C \rightarrow x_i \neq x_j$ so the resulting constraint is still a correct *ocond*. The last equivalence holds by replacing f by its definition and eliminating shared terms. Compared to the $\mathit{ocond}(\sigma)$ when we used f instead of f', there is an additional term $(p[x_{i-1}][x_j] + p[x_i][x_{j+1}] = p[x_{i-1}][x_i] + p[x_j][x_{j+1}] \wedge x_j < x_i$ which says that we can also prune the current assignment if the flipped version has *equal* objective value but a better lexicographical value for $\{x_1, \ldots, x_n\}$. Thus $\mathit{db}(\sigma_{i,j})$ not only breaks dominances but also includes a conditional symmetry breaking constraint. Similarly, consider $\sigma_{1,n}$. Because it is a boundary case, the terms in $\sigma(f)$ and f all cancel and we have $\mathit{ocond}(\sigma_{1,n}) \equiv x_n < x_1$, so $\mathit{db}(\sigma_{1,n}) \equiv x_1 \leq x_n$ which is simply a symmetry breaking constraint. □

Example 10 Consider the Knapsack problem from Example 4. In Step 0, we can tie break solutions with equal objective value by the weight used, and then lexicographically, i.e., $f' = \mathit{lex}(f, \sum w_i x_i, x_1, \ldots, x_n)$. In Step 3*, we have: $\forall i < j$, $\mathit{ocond}(\sigma_{i,j}) \equiv \sigma(f') < f' \Leftrightarrow ((v_i - v_j)(x_i - x_j) < 0) \vee ((v_i - v_j)(x_i - x_j) = 0 \wedge (w_i - w_j)(x_i - x_j) > 0) \vee ((v_i - v_j)(x_i - x_j) = 0 \wedge (w_i - w_j)(x_i - x_j) = 0 \wedge x_j < x_i)$. In Step 4, after simplifying, in addition to the dominance breaking constraints we had before, we would also have: $\mathit{db}(\sigma_{i,j}) \Leftrightarrow x_i \leq x_j$ if $w_i > w_j$ and $v_i = v_j$,

$db(\sigma_{i,j}) \Leftrightarrow x_i \geq x_j$ if $w_i < w_j$ and $v_i = v_j$, and $db(\sigma_{i,j}) \Leftrightarrow x_i \leq x_j$ if $w_i = w_j$ and $v_i = v_j$ which is a symmetry breaking constraint. $\square$

We can also apply the altered method to satisfaction problems to generate symmetry and conditional symmetry breaking constraints.

Example 11 The Black Hole Problem [17] seeks to find a solution to the Black Hole patience game. In this game the 52 cards of a standard deck are laid out in 17 piles of 3, with the Ace of spades starting in a "black hole". Each turn, a card at the top of one of the piles can be played into the black hole if it is numbered ± 1 from the number of the card that was played previously, with king wrapping back around to ace. The aim is to play all 52 cards. We can model the problem as follows. Let the suits be numbered from 1 to 4 in the order spades, hearts, clubs, diamonds. Let the cards be numbered from 1 to 52 so that card i has suit $(i-1)/13+1$ and number $(i-1) \bmod 13+1$, where 11 is jack, 12 is queen and 13 is king. Let $l_{i,j}$ be the jth card in the ith pile in the initial layout. Let x_i be the turn in which card i was played. Let y_i be the card which was played in turn i. We have:

$$x_1 = 1 \tag{1}$$
$$inverse(x, y) \tag{2}$$
$$x_{l_{i,j}} < x_{l_{i,j+1}} \qquad \forall 1 \leq i \leq 17, 1 \leq j \leq 2 \tag{3}$$
$$(y_{i+1} - y_i) \bmod 13 \in \{-1, 1\} \qquad \forall 1 \leq i \leq 51 \tag{4}$$

We now apply our method. Since cards which are nearer to the top of the piles are much more likely to be played early on, we choose a lexicographical ordering which reflects this. We define $f' = lex(x_{l_{1,1}}, \ldots, x_{l_{17,1}}, \ldots, x_{l_{1,3}}, \ldots, x_{l_{17,3}})$. An obvious set of mappings that are likely to map solutions to solutions is to swap cards of the same number in the sequence of cards to be played. Consider $\sigma_{i,j}$ for $i - j \bmod 13 = 0, i \neq 1, j \neq 1$ where $\sigma_{i,j}$ swaps x_i and x_j, and swaps the values of i and j among $\{y_1, \ldots, y_{52}\}$.

Now we construct $scond(\sigma_{i,j})$. For each constraint c in the problem, we need to find a c' such that $C \wedge D \wedge c' \Rightarrow \sigma_{i,j}(c)$ and add it to $scond(\sigma_{i,j})$. Clearly, the domain constraints and the constraints in (1), (2) and (4) are all symmetric in $\sigma_{i,j}$, so we do not need to add anything for them. However, there will be some constraints in (3) which are not symmetric in $\sigma_{i,j}$. For example, suppose we wished to swap $i = 3(3\spadesuit)$ and $j = 16(3\heartsuit)$, and they were in piles: $(2\spadesuit, 3\spadesuit, 5\clubsuit)$ and $(1\diamondsuit, 3\heartsuit, 6\diamondsuit)$, where $3\spadesuit$ is in a lexicographically earlier pile than $3\heartsuit$. The constraints in (3) which are not symmetric in $\sigma_{i,j}$ are those involving $3\spadesuit$ or $3\heartsuit$, i.e., $x_{2\spadesuit} < x_{3\spadesuit}$, $x_{3\spadesuit} < x_{5\clubsuit}$, $x_{1\diamondsuit} < x_{3\heartsuit}$ and $x_{3\heartsuit} < x_{6\diamondsuit}$. Their symmetric versions are $x_{2\spadesuit} < x_{3\heartsuit}$, $x_{3\heartsuit} < x_{5\clubsuit}$, $x_{1\diamondsuit} < x_{3\spadesuit}$ and $x_{3\spadesuit} < x_{6\diamondsuit}$ respectively, so we can set $scond(\sigma_{3\spadesuit,3\heartsuit}) \equiv x_{2\spadesuit} < x_{3\heartsuit} \wedge x_{3\heartsuit} < x_{5\clubsuit} \wedge x_{1\diamondsuit} < x_{3\spadesuit} \wedge x_{3\spadesuit} < x_{6\diamondsuit}$. To construct $ocond(\sigma_{i,j})$, we can set $ocond(\sigma_{i,j}) \equiv \sigma_{i,j}(f') < f'$. For this example, we have $ocond(\sigma_{3\spadesuit,3\heartsuit}) \equiv x_{3\heartsuit} < x_{3\spadesuit}$. Combining, we have $db(\sigma_{3\spadesuit,3\heartsuit}) \equiv \neg(x_{2\spadesuit} < x_{3\heartsuit} \wedge x_{3\heartsuit} < x_{5\clubsuit} \wedge x_{1\diamondsuit} < x_{3\spadesuit} \wedge x_{3\spadesuit} < x_{6\diamondsuit} \wedge x_{3\heartsuit} < x_{3\spadesuit})$. We can use the constraints in the original problem to simplify this further. Since $x_{3\spadesuit} < x_{5\clubsuit}$ is an original constraint and $x_{3\heartsuit} < x_{3\spadesuit} \wedge x_{3\spadesuit} < x_{5\clubsuit} \Rightarrow x_{3\heartsuit} < x_{5\clubsuit}$, we can eliminate the second term in $db(\sigma_{3\spadesuit,3\heartsuit})$. Since $x_{1\diamondsuit} < x_{3\heartsuit}$ is an original constraint and $x_{1\diamondsuit} < x_{3\heartsuit} \wedge x_{3\heartsuit} < x_{3\spadesuit} \Rightarrow x_{1\diamondsuit} < x_{3\spadesuit}$, we can eliminate the third term in $db(\sigma_{3\spadesuit,3\heartsuit})$. The result is $db(\sigma_{3\spadesuit,3\heartsuit}) \Leftrightarrow \neg(x_{2\spadesuit} < x_{3\heartsuit} \wedge x_{3\spadesuit} < x_{6\diamondsuit} \wedge x_{3\heartsuit} < x_{3\spadesuit})$. The other cases are similar. $\square$

Although the conditional symmetry breaking constraints derived in Example 11 are identical to those derived in an earlier paper on conditional symmetry breaking [17], our method is much more generic and can be applied to other problems as well. Also, no rigorous proof of correctness is given in that paper, whereas Theorem 3 shows that these conditional symmetry breaking constraints are compatible. In this problem it is quite possible to derive multiple incompatible conditional symmetry breaking constraints which are individually correct. For example, suppose in addition to $(2\spadesuit, 3\spadesuit, 5\clubsuit)$ and $(1\diamondsuit, 3\heartsuit, 6\diamondsuit)$, we had a third pile $(2\heartsuit, 3\diamondsuit, 7\spadesuit)$, then the following conditional symmetry breaking constraints are all individually correct: $\neg(x_{2\spadesuit} < x_{3\heartsuit} \wedge x_{3\spadesuit} < x_{6\diamondsuit} \wedge x_{3\heartsuit} < x_{3\spadesuit})$, $\neg(x_{2\heartsuit} < x_{3\spadesuit} \wedge x_{3\diamondsuit} < x_{5\clubsuit} \wedge x_{3\spadesuit} < x_{3\diamondsuit})$, $\neg(x_{1\diamondsuit} < x_{3\diamondsuit} \wedge x_{3\heartsuit} < x_{7\spadesuit} \wedge x_{3\diamondsuit} < x_{3\heartsuit})$, but they are incompatible. For example, no matter which permutation of $3\spadesuit$, $3\heartsuit$, and $3\diamondsuit$ is applied, the partial solution $1\spadesuit, 2\heartsuit, 1\diamondsuit, 2\heartsuit, 3\spadesuit, 4\spadesuit, 3\heartsuit, 4\diamondsuit, 3\diamondsuit$ is pruned by one of the three conditional symmetry breaking constraints. Our method will never produce such incompatible sets of dominance breaking constraints.

Example 12 The Resource Constrained Project Scheduling Problem (RCPSP) [4] is as follows. We have n tasks and m renewable resources. Each task i has a duration p_i and consumes $r_{i,j}$ units of resource j per time unit during its execution. Each resource i supplies a constant amount R_i of resource per time unit during the planning period. There are precedence constraints between certain pairs of tasks. The problem is to minimize the makespan of the schedule subject to the resource constraints and precedence constraints. Let $s_i \in \{0, \ldots, T\}$ be the start time of task i where T is the scheduling horizon. Let P be the set of precedences. Then the problem can be stated as follows:

$$\begin{array}{lll}
\text{Minimize} & max(s_i + p_i) & \\
\text{Subject to} & s_i + p_i \leq s_j & \forall (i,j) \in P \\
 & cumulative(s, p, [r_{j,i} \mid 1 \leq j \leq n], R_i) & \forall 1 \leq i \leq m
\end{array}$$

A well known dominance rule for this problem is that each task must start at the end time of another task, otherwise, it can be shifted forward in time for a possibly better solution. We show that it is straightforward to derive a dominance breaking constraint for this using our method. Let $f' = lex(f, s_1, \ldots, s_n)$, i.e., we tie break the objective function by the start times of the tasks, preferring schedules where they start earlier. Consider a mapping σ_i, where we take task i and shift it one time unit earlier. Clearly, $f(\sigma_i) < f$ is always true, so we can set $ocond(\sigma_i) \Leftrightarrow true$.

Now we construct $scond(\sigma_i)$. For each constraint c in the problem, we need to find a c' such that $C \wedge D \wedge c' \Rightarrow \sigma_i(c)$ and add it to $scond(\sigma_i)$. Suppose our current solution is θ and task i starts at s_i. σ_i maps s_i to $s_i - 1$. The domain constraints are all symmetric in σ_i except for those on s_i. We have $\sigma_i(s_i \geq 0) \equiv s_i \geq 1$. Since we already have $s_i \geq 0$, we can simply add $c' \equiv s_i \neq 0$ to $scond(\sigma_i)$. Consider the cumulative constraint for resource k. If task i does not use resource k, then $\sigma_i(\theta)$ is always a solution of c. If task i does use resource k, then a sufficient condition for $\sigma_i(\theta)$ to be a solution of c is that none of the tasks which use resource k end at exactly s_i. So we can set $c' \equiv \wedge_{\{j \mid r_{j,k} > 0\}} s_i \neq s_j + p_j$. Now consider one of the precedence constraints. If it does not involve task i, or if task i is the predecessor, then $\sigma_i(\theta)$ is always a solution of c. If task i is the successor, and task j is the predecessor, then $\sigma_i(\theta)$ is a solution of c iff $s_j + p_j \neq s_i$. So we can set $c' \equiv s_i \neq s_j + p_j$. Let T_i be the set of tasks which either share a resource with task

i, or is a predecessor of task i. Then, collecting all the terms together, we have: $scond(\sigma_i) \equiv s_i \neq 0 \wedge \wedge_{j \in T_i} s_i \neq s_j + p_j$. So:

$$db(\sigma_i) \equiv \neg(scond(\sigma_i) \wedge ocond(\sigma_i)) \Leftrightarrow s_i \in \{0\} \cup \{s_j + p_j \mid j \in T_i\}.$$

This is simply an element constraint and can be implemented in a straightforward manner. □

Example 13 The Nurse Scheduling Problem (NSP) is to schedule a set of nurses over a time period such that work and hospital regulations are all met, and as many as possible of the nurses' preferences are satisfied. There are many variants of this problem in the literature (e.g., [27,1]). We pick a simple variant to illustrate our method. Each day has three shifts: day, evening, and overnight. On each day, each nurse should be scheduled into one of the three shifts or scheduled a day off. For simplicity, we can consider a day off to be a shift as well. We number the shifts as day: 1, evening: 2, over-night: 3, day-off: 4. Each shift besides day-off requires a minimum number r_i of nurses to be rostered. Nurses cannot work for more than 6 days in a row, and must work at least 10 shifts per 14 days. Each nurse i has a preference $p_{i,j}$ for which of the four shifts they wish to take on day j. The objective is to maximize the number of satisfied preferences. Let n be the number of nurses and m be the number of days. Let $x_{i,j}$ be the shift that nurse i is assigned to on day j. Then the problem can be stated as follows:

$$\text{Maximize} \quad \sum_{i=1}^{n}\sum_{j=1}^{m}(x_{i,j} = p_{i,j})$$

Subject to

$$at\_least([x_{k,j} \mid 1 \leq k \leq n], i, r_i) \qquad \forall 1 \leq i \leq 3, 1 \leq j \leq m \qquad (5)$$

$$at\_least([x_{i,j} \mid k \leq j < k+7], 4, 1) \qquad \forall 1 \leq i \leq n, 1 \leq k \leq n-6 \qquad (6)$$

$$at\_most([x_{i,j} \mid k \leq j < k+14], 4, 4) \qquad \forall 1 \leq i \leq n, 1 \leq k \leq n-13 \qquad (7)$$

We now apply our dominance breaking method. Firstly, we can potentially get some symmetry or conditional symmetry breaking in by refining the objective function to $f' = lex(f, x_{1,1}, x_{2,1}, \ldots, x_{n,m})$. Let us consider mappings which are likely to map solutions to solutions. An obvious set of candidates are mappings which swap the shifts of two nurses on the same day, i.e., mappings $\sigma_{i_1,i_2,j}$ which swap $x_{i_1,j}$ and $x_{i_2,j}$.

We wish to calculate $scond(\sigma_{i_1,i_2,j})$. For each $c \in C \cup D$, we need to find c' such that $C \wedge D \wedge c' \Rightarrow \sigma_{i_1,i_2,j}(c)$. It is easy to see that the constraints in (5) are all symmetric in $\sigma_{i_1,i_2,j}$, so we do not need to add anything to $scond(\sigma_{i_1,i_2,j})$. The constraints $at\_least([x_{i_1,j'} \mid k \leq j' < k+7], 4, 1)$ in (6) are symmetric in $\sigma_{i_1,i_2,j}$ if $k + 7 \leq j$ or $k > j$. For $j - 7 < k \leq j$, they will be satisfied by $\sigma(\theta)$ iff: $x_{i_1,j} \neq 4 \vee x_{i_2,j} = 4 \vee at\_least([x_{i_1,j'} \mid k \leq j' < k+7, j \neq j'], 4, 1)$. Similarly, the constraints $at\_most([x_{i_1,j} \mid k \leq j < k+14], 4, 4)$ in (7) are symmetric in $\sigma_{i_1,i_2,j}$ if $k + 14 \leq j$ or $k > j$. For $j - 14 < k \leq j$, they will be satisfied by $\sigma(\theta)$ iff: $x_{i_1,j} = 4 \vee x_{i_2,j} \neq 4 \vee at\_most([x_{i_1,j'} \mid k \leq j' < k+14, j \neq j'], 4, 3)$. Since the *at_least* and *at_most* conditions are probably too expensive to check, we can simply throw them away. We lose some potential pruning, but it is still correct, since we had a disjunction of conditions. So we add $x_{i_1,j} \neq 4 \vee x_{i_2,j} = 4$ and $x_{i_1,j} = 4 \vee x_{i_2,j} \neq 4$ to $scond(\sigma_{i_1,i_2,j})$. Calculating $\sigma(f') > f'$ is straightforward. We simply try each pair of values for $x_{i_1,j}$ and $x_{i_2,j}$ and see if swapping them improves the refined objective function. For example, suppose $i_1 < i_2$, $p_{i_1,j} = 4$, $p_{i_2,j} = 2$, then

$$ocond(\sigma_{i_1,i_2,j}) \equiv \sigma(f') < f' \Leftrightarrow (x_{i_1,j}, x_{i_2,j}) \in \{(1,4),(2,1),(2,3),(2,4),(3,1),(3,4)\}.$$

And finally

$$db(\sigma_{i_1,i_2,j}) \equiv \neg(scond(\sigma_{i_1,i_2,j}) \wedge ocond(\sigma_{i_1,i_2,j})) \Leftrightarrow (x_{i_1,j}, x_{i_2,j}) \notin \{(2,1),(2,3),(3,1)\},$$

which is a table constraint encapsulating both dominance breaking and symmetry breaking constraints. □

5 Interaction with Search

It is well known that static symmetry breaking constraints can conflict with the search heuristic [16], leading to little speedup or even an overall slow down. Since dominance breaking constraints are a direct generalization of symmetry breaking constraints, the same issues are pertinent. Clearly, if we refine the objective function using a lexicographical ordering which is contradictory to the order in which the search strategy would like to search, the symmetry breaking and conditional symmetry breaking constraints defined by our method will conflict with the search strategy in the same way that symmetry breaking constraints does in static symmetry breaking. The solution in both cases is to pick a lex ordering that is as closely aligned to the search strategy as possible. For example, if the search is a fixed order search on $[v_1, \ldots, v_n]$ where the value ordering is from smallest to largest, then $f' = lex(f, v_1, \ldots, v_n)$ is a good refinement. If the search tries values from largest to smallest, then we should use $f' = lex(f, -v_1, \ldots, -v_n)$. If the search uses a dynamic variable ordering, then it may not be possible to pick a refinement which is completely consistent with the search strategy.

While the symmetry and conditional symmetry breaking constraints defined by our method may suffer from the already well known conflict with the search strategy described above, a new possibility for conflict arises between the exploitation of strict dominances and the branch and bound framework often used for solving optimization problems. Dominance breaking constraints attempt to prevent the solver from searching any dominated subtree whatsoever. However, while dominated solutions may not be optimal, they may nevertheless allow additional pruning in a branch and bound framework if they improve the current best solution. Consider a situation where we have a relatively bad search heuristic. It may have ordered the search tree such that the first 1000 solutions it encounters are all bad, dominated solutions, and it does not encounter a good, non-dominated one until the 1001st one. In contrast, a search without dominance breaking constraints will find these bad solutions and use them for branch and bound. So while it does not benefit from the additional pruning of the dominance breaking constraints, it has compensation in the form of a better bound for performing branch and bound in that part of the search. A search with dominance breaking constraints on the other hand, will enjoy the additional pruning from dominance breaking constraints, but will never find any of those first 1000 dominated solutions. This means that for that stretch of search, it had no bound on the objective with which to perform branch and bound and is missing out on some potential pruning. The end result is that it may spend a large amount of time with no solution at all, although once it does find a solution, it tends to be a very good one. In pathological cases, it is entirely possible that with dominance breaking constraints, the solver will not find any solution within the time out at all, whereas without dominance breaking constraints, it will at least find some bad solutions. Thus dominance breaking constraints are not necessarily beneficial, especially in an any-time context where you want to find good solutions fast. We illustrate this kind of conflict experimentally in Section 7.

Note that this new type of possible conflict between exploiting strict dominances and the branch and bound framework is fundamentally different from the possible conflict between lexicographical symmetry breaking constraints and the search strategy. In the latter type of conflict, the conflict is caused by the user artificially deciding which solution among a set of equally good, fully symmetric solutions to accept and which to prune, such that that choice conflicts with the order in which the search normally finds those solutions. Such a conflict can be resolved by using a good lexicographical ordering which is consistent with the search strategy, by dynamically changing the lexicographical ordering during search [20], or by using dynamic symmetry breaking techniques such as SBDS [3, 18] or SBDD [10,14] which determine the ordering dynamically. However, the new kind of conflict identified here is caused by the order between pairs of dominated solutions where one has a strictly better objective value than the other. If the search tends to find the better one of each pair of dominated solutions first, it will tend not to conflict with the strict dominance breaking. On the other hand, if it consistently finds the worse one of each pair first, it will tend to conflict with the dominance breaking. This is because the dominance breaking constraints may prune off this worse solution that it encounters first, even though it is possible that this solution is better than the current best and can help the branch and bound to prune more. Thus the conflict is caused directly by the search strategy ordering the solutions badly, and is not caused by some inappropriate choice of ordering by the user when generating the dominance breaking constraints. Changing the lexicographical ordering used for the refinement phase either statically or dynamically will not fix this conflict. Nor will any direct extension of SBDS or SBDD that we can think of. Instead, the best way to avoid the conflict is simply to use a good search strategy, since a good search strategy will order the good solutions first and will tend not to conflict with the dominance breaking.

If a good search strategy cannot be found, an alternative method called *dominance jumping* [8] can be used to resolve the conflict instead. The basic idea behind dominance jumping is that whenever the current subtree is pruned by a dominance breaking constraint $db(\sigma)$, we actually know exactly where the better subtree that dominates the current one is. If D is the domain of the current subtree, and $db(\sigma)$ prunes the current subtree (every solution of θ in D is dominated by $\sigma(\theta)$), then $\sigma(D)$[2] leads to a subtree that dominates the current one. The dominance jumping method modifies the search so that instead of simple failing when $db(\sigma)$ is violated and backtracking and continuing with the depth first search, the search immediately jumps to the subtree reached by $\sigma(D)$. What this means is that even if the search is bad and has ordered a dominated solution first, once that dominated subtree is encountered, the search will simply jump to the better subtree containing a dominating solution. Thus we can find those dominating solutions quickly and enjoy the full benefits of branch and bound while still avoiding searching any dominated subtrees.

[2] The dominance jumping method relies on σ being extendible to map domains to domains, which is true for almost all σ considered in practice.

One might think that if a good search heuristic is used, then dominance breaking constraints may be useless, because the search never gets to the dominated solutions in the first place. However, this is not true. If a good search heuristic is used, the first parts of the search tree that are explored may contain few or no dominated solutions, meaning that dominance breaking constraints provide little benefit there. However, a complete search must eventually also search the bad parts of search tree to prove that no better solution exists. These parts of the search tree may contain lots of dominated solutions, and dominance breaking constraints can be highly useful there, as they provide a completely complementary pruning scheme to branch and bound, and can often prove that a bad subtree is bad exponentially faster than pure branch and bound can. We illustrate this with the following example.

Example 14 In the knapsack problem, suppose we have $v_1 = 2$, $w_1 = 1$, $v_i = 1, w_i = 1$ for $i = 2, \ldots, 100$ and $W = 50$. Any decent search heuristic will quickly lead us to an optimal solution of profit 51. However, even after the optimal solution is found, branch and bound cannot immediately detect that the partial assignment $x_1 = 0$ is suboptimal (unless we use a global propagator like a knapsack propagator or a linear programming propagator). In fact, it will still spend an exponential amount of time down the $x_1 = 0$ branch to prove no solution better than 51 exists. However, the dominance constraints will enforce $x_1 \geq x_i$ for $i = 2, \ldots, 100$, so as soon as we try $x_1 = 0$, propagation will detect failure.

In general, dominance constraints can allow us to detect local suboptimalities and prune a subtree even if that suboptimality is not enough to immediately make the bound on the objective value sufficiently bad for branch and bound to prune it.

6 Related Work

There have been many works on problem specific applications of dominance relations, e.g., the template design problem [33], online scheduling problems [19], the Maximum Density Still Life problem, Steel Mill Design problem and Peaceable Armies of Queens problem [32], the Minimization of Open Stacks problem [7], and the Talent Scheduling Problem [15]. However, the methods used are typically highly problem specific and offer little insight as to how they can be generalized and applied to other problems. The implementations of these methods are also often quite ad-hoc (e.g., pruning values from domains even though they do not explicitly violate any constraint), and it is not clear whether they can be correctly combined with other constraint programming techniques, such as restarts or nogood learning. Many of these methods alter the search in order to implement dominance breaking. They can be seen as performing a somewhat non-rigorous propagation of a dominance breaking constraint directly in the search engine in order to remove possible values for the next decision variable, rather than a proper propagation of a dominance breaking constraint in the propagation engine. For example, in the Photo problem, instead of propagating $db(\sigma_{i,j}) \equiv p[x_{i-1}][x_j] + p[x_i][x_{j+1}] \leq p[x_{i-1}][x_i] + p[x_j][x_{j+1}]$ in the propagation engine, they would not propagate it, but would wait till they are about to label x_{j+1} and then use those dominance breaking constraints to figure out which values of x_{j+1} do not need to be searched. The result is a solver that is not very rigorous because it is technically no longer a complete search. The dominance breaking constraint used is rarely explicitly stated, and it is rarely formally proved that such solvers are actually correct. This is particularly problematic when multiple dominances and symmetries are being exploited simultaneously, as then the correctness of the solver is not obvious at all. In contrast, our new method rests on a much stronger theoretical foundation and is completely rigorous. Since our method simply adds constraints to the problem, the modified problem is a perfectly normal constraint problem and it is correct to use any other constraint programming technique on it. Another important advantage of our method is that we are able to use any search strategy we want on the modified problem. This is not the case with many of the problem specific dominance breaking methods as they rely on specific search orders.

There are a small number of works on generic methods for detecting and exploiting dominance relations. Machine learning techniques have been proposed as a method for finding candidate dominance relations [38]. This method works by encoding problems and candidate dominance relations into forms amenable to machine learning. Machine learning techniques such as experimentation, deduction and analogy are then used to identify potential dominance relations. This method was able to identify dominance relations for the 0/1 knapsack problem and a number of scheduling problems. However, the main weakness of this method is that it only generates candidate dominance relations and does not prove their correctness. Each candidate has to be analyzed to see if it is in fact a dominance relation. Then the dominance relation has to be manually proved and exploited.

Recently, several generic and automatic methods have been developed for exploiting certain classes of dominance relations. These include nogood learning techniques such as Lazy Clause Generation [30,11] and Automatic Caching via Constraint Projection [6]. Both of these can be thought of as dynamic dominance breaking, where after some domain D_1 is found to fail, a nogood (constraint) n is found which guarantees that if D_2 violates n, then D_2 is dominated by D_1 and must also fail. The nogood n is posted as an additional redundant constraint to the problem. Lazy Clause Generation derives this n by resolving together clauses which explain the inferences which led to the failure. Automatic Caching via Constraint Projection derives n by finding conditions such that projection of the subproblem onto the subset of unfixed variables yield a more constrained problem. These methods are to a large extent complementary to the method presented in this paper. None of these methods exhausts all possible dominances occurring in a problem, and there are dominances which can be exploited by one method but not another. Thus we can often use them simultaneously to gain an even greater reduction in search space.

7 Experimental Results

In this section, we present experimental results showing the utility and also the limitations of dominance breaking constraints.

7.1 The Utility of Dominance Breaking Constraints

We now give some experimental results for our method on a variety of problems. Note that the aim of our method is to accelerate the solving of an arbitrary model, not necessarily that of the best model, hence improving the state of the art on these problems is not the current aim. We compare using no dominance breaking or symmetry breaking constraints (base), with using symmetry breaking constraints only (sym), and using the dominance breaking constraints defined by our new method (dom). Note that the dominance breaking constraints defined by our method exploit full symmetries, conditional symmetries and strict dominances and hence are a superset of the symmetry breaking constraints. Note also that many problems only have conditional symmetries or strict dominances and do not exhibit any full symmetries. We have already discussed how our approach to dominance breaking applies to the Photo Problem, Knapsack Problem, Black Hole Problem, and Nurse Scheduling. For these problems, we generate random instances of several different sizes, with 10 instances of each size. We also give experimental results for four further problems:

Photo Problem. This problem has full symmetries, conditional symmetries and strict dominances as described in Example 3 and 9. We use a search strategy where we label the x_i in order. To label x_i, we try the available value with the highest $p[x_{i-1}][x_i]$ first.

Black Hole Problem. This problem has conditional symmetries as described in Example 11, but no full symmetries or strict dominances. We use a search strategy where out of the legally playable cards, we pick the one that is in the largest pile.

Knapsack Problem. This problem has symmetries and strict dominances as described in Example 4 and 10. We use a search strategy where we pick the unfixed x_i with the highest v_i/w_i and set it to 1 first.

Nurse Scheduling Problem. This problem has conditional symmetries and strict dominances as described in Example 13. It can also have instance specific full symmetries if for example two nurses have exactly the same preferences for all days in the scheduling period, in which case they become interchangeable. However, this does not occur in any of our instances. We use a search strategy where we label day by day, nurse by nurse within each day, and try to assign each nurse to the shift they most prefer.

RCPSP. The resource constrained project scheduling problem (RCPSP) [4] schedules n tasks using m renewable resources so that ordering constraints among tasks hold and resource usage limits are respected. A standard dominance rule for this problem, used in search, is that each task must start at time 0 or when another task ends, since any schedule not following this rule is dominated by one constructed by shifting tasks earlier until the rule holds. We use the instances from the standard J60 benchmark set [34] which are non-trivial (not solved by root propagation) and solvable by at least one of the methods we compare. This problem has conditional symmetries and strict dominances. If a task can be shifted forward in the schedule, but it does not reduce the makespan, then this is a conditional symmetry. If shifting it forward does reduce the makespan, then it is a strict dominance. There may also be instance specific full symmetries if for example two tasks have exactly the same resource requirements, duration and precedence constraints. However, no such instance specific symmetries occur in the J60 benchmarks we are using. We use a search strategy where we find the unfixed task with the earliest possible start time and set its start time to that earliest time.

Talent Scheduling Problem. In the Talent Scheduling Problem [15], we have a set of scenes and a set of actors. Each actor appears in a number of scenes and is paid a certain amount per day they are on location. They must stay on location from the first scene they are in till the last scene they are in. The aim is to find the schedule of scenes $x_1, \ldots, x_n$ which minimize the cost of the actors. We set $f' = lex(f, x_1, \ldots, x_n)$. We consider mappings which take one scene and move it to another position in the sequence. We generate 10 random instances of size 14, 16, and 18 scenes and 8 actors. This problem has conditional symmetries and strict dominances. Moving a scene to another position in the sequence may increase, maintain or decrease the total cost of hiring the actors. If it maintains the cost, it is a conditional symmetry. If it increases or decreases the cost, it is a strict dominance. There may also be instance specific full symmetries if for example two scenes require the exact same set of actors, in which case swapping the position of those scenes is a full symmetry. However, no such instance specific symmetries occur in our instances. We use a search strategy where we label day by day, and we pick the scene with the lowest score where the score is calculated as follows. For each actor who is on-site but not in the scene, we add the actor's cost to the score. For each actor who is not on-site but is in the scene, we add the actor's cost to the score.

Steel Mill Problem. In the Steel Mill Problem (CSPLIB problem number 38, originally presented in Kalagnanam et al. [23]), we have a set of orders to be fulfilled and the aim is to minimize the amount of wasted steel. Each order i has a size and a color (representing which path it takes in the mill) and is to be assigned to a slab x_i. Each slab can only be used for orders of two different colors. Depending on the sum of the sizes of the orders on each slab, a certain amount of steel will be wasted. We set $f' = lex(f, x_1, \ldots, x_n)$ and try mappings where we take all orders of a certain color from one slab, and all orders of a certain color from another slab, and swap the slabs they are assigned to. We generate 10 random instances of size 40 and 50. This problem has full symmetries, conditional symmetries and strict dominances. All the slabs are interchangeable, leading to a full symmetry on the slabs. Conditional symmetries arise when the orders of a certain color from one slab has the same total size as the orders of a certain color from another slab, in which case they can be exchanged without changing the objective value. Strict dominances can occur when the orders of a certain color from one slab has a different total size than the orders of a certain color from another slab, in which case we may be able to swap them and use a smaller slab for one of them, leading to less wasted steel. We use a search strategy where we label the slabs one by one. For each slab, we first set its wastage to the lowest allowed value, and then we go through each order and try to add it to the slab.

PC Board Problem. In the PC Board Problem [25], we have $n \times m$ components of various types which need to be assigned to m machines. Each machine must be assigned exactly n components and there are restrictions on the sets of components that can go on the same machine. Each type of component gains a certain utility depending on which machine it is assigned to and the goal is to maximize the overall utility. We set $f' = lex(f, x_{1,1}, x_{1,2}, \ldots, x_{n,m})$ where $x_{i,j}$ is the type of component assigned to the jth spot on the ith machine. We consider mappings which swap two components on different machines. We generate 20 random instances of size 6×8. This problem has conditional symmetries and strict dominances. If two components can be swapped without violating the restrictions on which sets of

Table 6 Comparison of the original model and the model augmented with symmetry breaking and dominance breaking constraints

Problem	base		sym		dom	
	Time	Nodes	Time	Nodes	Time	Nodes
Photo-14	1.09	57773	0.76	43785	0.90	10967
Photo-16	8.38	441574	7.34	383242	4.00	43373
Photo-18	60.68	2828622	49.7	2320255	22.09	206507
Knapsack-20	**0.01**	340	**0.01**	322	**0.01**	15
Knapsack-30	0.17	46422	0.15	41386	**0.01**	91
Knapsack-50	602	1×10^8	605	1×10^8	**0.01**	684
Knapsack-100	900	1×10^8	900	1×10^8	**0.40**	54705
Black-hole	5.18	77542	5.18	77542	**0.08**	607
Nurse-15-7	900	9×10^7	900	9×10^7	900	8×10^7
Nurse-15-14	900	8×10^7	900	8×10^7	900	8×10^7
RCPSP	358.95	2779652	358.95	2779652	279.74	781399
Talent-Sched-14	1.66	39479	1.66	39479	0.42	10122
Talent-Sched-16	16.08	349704	16.08	349704	2.33	51993
Talent-Sched-18	252.05	5557959	252.05	5557959	13.88	299043
Steel-Mill-40	60.64	1×10^6	65.7	1×10^6	22.00	451636
Steel-Mill-50	379.21	7×10^6	384.18	7×10^6	231.95	3×10^6
PC-board	547.93	4×10^7	547.93	4×10^7	412.29	1×10^7

components can go on the same machine, then depending on where two components are currently assigned to, swapping them may increase, maintain or decrease the utility. If the utility remain the same, it is a conditional symmetry. If the utility increases or decreases, it is a strict dominance. We use a search strategy where we label one machine at a time. For each machine, we pick the allowed component which has the highest utility in that machine and assign it there.

The experiments were performed on Xeon Pro 2.4GHz processors using the CP solver Chuffed. For each set of benchmarks, we report the geometric mean of time taken in seconds and the number of failed nodes. Table 6 compares the original problem with no symmetry breaking or dominance breaking of any form (base), with symmetry breaking constraints only (sym), with dominance breaking constraints constructed using our method (dom) Note that since as described above, some of the problem classes only have strict dominances or conditional symmetries and has no full symmetries, for those problem classes, symmetry breaking constraints cannot be applied and base and sym will be identical. A timeout of 900 seconds was used. Table 7 shows the results when we use a learning solver on each of the variants of the problem: the original with Lazy Clause Generation (base+lcg), with symmetry breaking constraints and Lazy Clause Generation (sym+lcg) and with dominance breaking constraints and Lazy Clause Generation (dom+lcg). Fastest times and lowest node counts across both tables are shown in bold. All the instances tested are available in MiniZinc [29] format at `www.cs.mu.oz.au/~pjs/dominance/`.

By comparing dom with base and dom+lcg with base+lcg in Tables 6 and 7, it is clear that adding dominance breaking constraints can significantly reduce the search space on a variety of problems, leading to large speedups which tend to grow exponentially with problem size. By comparing dom with sym and dom+lcg with sym+lcg, we can see that dominance breaking is doing a lot more pruning than pure symmetry breaking. In many of these problems, there are few or no full symmetries to exploit, but there are many conditional symmetries or strict dominances which can be exploited for significant speedup using our dominance

Table 7 Comparison of the original model and the model augmented with symmetry breaking and dominance breaking constraints using learning

Problem	base+lcg		sym+lcg		dom+lcg	
	Time	Nodes	Time	Nodes	Time	Nodes
Photo-14	0.30	5791	**0.22**	4588	0.25	**1962**
Photo-16	6.49	44325	4.46	40221	**1.40**	**8960**
Photo-18	19.73	138926	15.66	117132	**6.25**	**24523**
Knapsack-20	**0.01**	336	**0.01**	318	**0.01**	**11**
Knapsack-30	0.85	45733	0.77	40770	**0.01**	**65**
Knapsack-50	900	2×10^7	900	2×10^7	**0.01**	**507**
Knapsack-100	900	1×10^7	900	1×10^7	1.05	**37571**
Black-hole	0.97	2767	0.97	2767	0.09	**347**
Nurse-15-7	1.72	55217	1.72	55217	**0.91**	**24258**
Nurse-15-14	483.29	7×10^6	483.29	7×10^6	**140.95**	$\mathbf{1 \times 10^6}$
RCPSP	**4.07**	**7890**	**4.07**	**7890**	32.84	32770
Talent-Sched-14	0.45	4983	0.45	4983	**0.27**	**3189**
Talent-Sched-16	3.71	27186	3.71	27186	**1.28**	**12336**
Talent-Sched-18	26.25	128810	26.25	128810	**4.28**	**31829**
Steel-Mill-40	16.31	75293	19.2	82932	**4.53**	**27225**
Steel-Mill-50	249.39	714451	250.8	646581	**32.24**	**129788**
PC-board	20.28	156933	20.28	156933	**7.51**	**64320**

breaking constraints. Although we only compared against symmetry breaking constraints here, other symmetry breaking methods such as SBDS [18] or SBDD [10] are also incapable of exploiting conditional symmetries or strict dominances (despite the word "dominance" appearing in the name of the SBDD method). The speedup here between the dominance breaking and symmetry breaking is caused by the exploitation of the conditional symmetries and strict dominances which no pure symmetry breaking method can exploit.

Dominance breaking constraints are also orthogonal to nogood learning techniques such as Lazy Clause Generation, and can be combined with it for additional speedup (e.g., Photo, Steel Mill, Talent Scheduling, Nurse Scheduling, PC Board). In some cases (e.g., Knapsack, Black Hole), even though adding LCG on top of our method can reduce the node count further, the extra overhead of LCG swamps out any benefit. In other cases (e.g., RCPSP), adding our dominance breaking constraints on top of LCG actually increases the run time and node count. In this problem, the dynamically derived dominances from LCG are stronger than the static ones that our method derives. Adding the dominance breaking constraints interferes with and reduces the benefit of LCG. In general however, our dominance breaking constraints appears to provide significant speedups over a wide range of problems for both non-learning and nogood learning solvers.

7.2 Conflict Between Dominance Breaking Constraints and Search

In the next set of experiments, we illustrate what happens when the dominance breaking constraints conflict with the search strategy as described in Section 5. We will use the Steel Mill problems of size 50 used in the previous experiment. However, instead of using a good search strategy, we are going to use increasingly bad search strategies to see what sort of interaction there is between the search and the dominance breaking constraints. A reasonably good search strategy is to label one slab at a time, and for each slab, first set its wastage variable to its lower bound, and then decide which orders to put on it. To make the search strategy

Table 8 Comparing the effectiveness of dominance breaking constraints as the search strategy goes from good to bad on the Steel Mill problem.

Search	opt. %		sat. %		sat. time		val.	
	base	dom	base	dom	base	dom	base	dom
follow-100%	20	**70**	**100**	**100**	**0.02**	0.13	2.9	**0.6**
follow-75%	20	**70**	**100**	**100**	**0.01**	11.2	10.8	**4.1**
follow-50%	**20**	**20**	**100**	90	**0.02**	51.8	28.5	**7.3**
follow-25%	**20**	10	**100**	70	**0.02**	93.7	44.6	**9.6**
follow-0%	**10**	**10**	**100**	70	**0.02**	261.6	71.1	**13.7**

worse, we can force it to pick some suboptimal values of the wastage variable to try first. Let follow-x% denote the search strategy where we pick a value at the xth percentile of goodness for the wastage variables. So for example, follow-100% will try setting the wastage variable to its lower bound first, follow-0% will try to set the wastage variable to its upper bound first, and follow-50% will try to set it to the median value in its domain first, etc. Thus the search strategy gets increasingly worse as the follow percentage drops. We compare using no symmetry breaking or dominance breaking (base) with using dominance breaking (dom). The lexicographical ordering used for the symmetry breaking part of the dominance breaking constraints is chosen so that it does not conflict with the search strategy. Thus any conflict that occurs is due to the interaction between the search and the strict dominance part of the dominance breaking constraints. We show the proportion of instances solved to optimality (opt. %), the proportion of instances where at least one solution was found (sat. %), the geometric mean of the time to find the first solution (sat. time) and the arithmetic mean of the best solution found among the instances where at least one solution was found (val.).

It can be seen from Table 8 that when a good search strategy is used, dominance breaking constraints are highly effective, allowing many more instances to be solved to optimality. However, as the search strategy gets worse, several things occur. First, it takes longer and longer for a first solution to be found when using dominance breaking constraints, whereas without dominance breaking constraints, the time to find the first solution is pretty much a constant 0.02 seconds or so. This slowdown in finding the first solution is due to the conflict between the dominance breaking constraints and the search strategy, as described in Section 5. Secondly, when the search strategy is sufficiently bad (at around follow-50%), it can take so long to find a first solution with dominance breaking constraints that it sometimes does not actually manage to find one at all within the time out. Thirdly, although it takes an increasingly longer time to find a first solution with dominance breaking constraints, if it does find one, it is typically of much higher quality than the ones found without dominance breaking constraints.

8 Conclusion and Future Work

We have described a generic method for identifying and exploiting dominance relations in constraint problems. The method defines a set of dominance breaking constraints which are provably correct and compatible with each other. The method also defines symmetry and conditional symmetry breaking constraints as a special case, thus it unifies symmetry breaking, conditional symmetry breaking and dominance breaking under one method. Experimental results show that the dominance breaking constraints we define can lead to significant reductions in search space and run time on a variety of problems, and that they can be effectively combined with other dominance breaking techniques such as Lazy Clause Generation.

Although we have developed this method in the context of Constraint Programming, the dominance relations we find can be applied to other kinds of search as well. For example, MIP solvers, which use branch and bound, can also benefit from the power of dominance relations, as they can encounter suboptimal partial assignments which nevertheless do not produce an LP bound strong enough to prune the subproblem. Simple dominance rules such as fixing a variable to its upper/lower bound if it is only constrained from below/above [21] are already in use in MIP, but our method can produce much more generic dominance rules. Similarly, local search can benefit tremendously from dominance relations, as they can show when a solution is suboptimal and map it to another solution which is better. Exploring how our method could be adapted for use in other kinds of search is an interesting avenue of future work.

It may also be possible to automate many or all of the steps involved in our method. Such automation would provide a great benefit for system users as they will be able to feed in a relatively "dumb" model and have the system automatically identify and exploit the dominances. Step 0 typically requires augmenting the objective function with an appropriate lexicographical ordering of the variables. Simple methods such as ordering the variables based on the order they are created or based on the order they are labelled in the search work well. For Step 1, Table 2 gives a list of standard σ's we can try. There also exist automated methods for detecting symmetries in problem instances [26,35] which could be adapted to look for additional candidates for σ. Step 2 and 3 involve algebraic manipulations which are not difficult for a computer to do. Assuming that all constraints have been annotated with any functional or monotonic properties, it is straightforward to apply the rules contained in Table 3, 4 and 5 to derive candidates for $scond(\sigma)$ and $ocond(\sigma)$. The difficulty lies in choosing whether to use simplified forms of $scond(\sigma)$ and $ocond(\sigma)$ if they are available, as there is a tradeoff between speed and pruning and there may not be a clear winner. We could either go for a default (e.g., always pick simplest), or present the options to a human, who can then choose. Another difficulty lies in Step 4, where we need to simplify the dominance breaking constraint and determine whether it is sufficiently simple, efficient and powerful that it is worth adding to to problem. This could potentially be done via some some of automated empirical testing where we initially add it, but monitor whether it is actually pruning anything. If not, we can disable it to save on overhead. Automating the method is another interesting avenue of future work.

Acknowledgments

NICTA is funded by the Australian Government as represented by the Department of Communications and the Australian Research Council. This work was partially supported by Asian Office of Aerospace Research and Development (AOARD) Grant FA2386-12-1-4056.

References

1. Slim Abdennadher and Hans Schlenker. Nurse scheduling using constraint logic programming. In S. Uthurusamy and B. Hayes-Roth, editors, *Proceedings of the Innovative*

Applications iof Artificial Intelligence Conference, pages 838–843, 1999.

2. Tariq A. Aldowaisan. A new heuristic and dominance relations for no-wait flowshops with setups. *Computers & OR*, 28(6):563–584, 2001.
3. Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming CP1999*, volume 1713 of *LNCS*, pages 73–87. Springer, 1999.
4. Peter Brucker, Andreas Drexl, Rolf H. Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.
5. G. Chu and P.J. Stuckey. A generic method for systematically identifying and exploiting dominance relations. In *Proceedings of the 18th International Conference on Principles and Practice of Constraints Programming CP2012*, number 7514 in LNCS, pages 6–22. Springer, 2012.
6. Geoffrey Chu, Maria Garcia de la Banda, and Peter J. Stuckey. Automatically exploiting subproblem equivalence in constraint programming. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *LNCS*, pages 71–86. Springer, 2010.
7. Geoffrey Chu and Peter J. Stuckey. Minimizing the maximum number of open stacks by customer search. In Ian P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraints Programming*, volume 5732 of *LNCS*, pages 242–257. Springer, 2009.
8. Geoffrey Chu and Peter J. Stuckey. Dominance driven search. In C. Schulte, editor, *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, volume 8124 of *LNCS*, pages 217–229. Springer, 2013.
9. James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.
10. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *LNCS*, pages 93–107. Springer, 2001.
11. Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *Proceedings of the 15th International Conference on Principles and Practice of Constraints Programming CP2009*, volume 5732 of *LNCS*, pages 352–366. Springer, 2009.
12. Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking Row and Column Symmetries in Matrix Models. In Pascal Van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraints Programming CP2002*, volume 2470 of *LNCS*, pages 462–476. Springer, 2002.
13. Pierre Flener, Justin Pearson, Meinolf Sellmann, and Pascal Van Hentenryck. Static and dynamic structural symmetry breaking. In *Proceedings of the 12th International Conference on Principles and Practice of Constraints Programming CP2006*, pages 695–699. Springer, 2006.
14. Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *LNCS*, pages 77–92. Springer, 2001.
15. Maria Garcia de la Banda, Peter J. Stuckey, and Geoffrey Chu. Solving talent scheduling with dynamic programming. *INFORMS Journal on Computing*, 23(1):120–137, 2011.
16. Antoine Gargani and Philippe Refalo. An efficient model and strategy for the steel mill slab design problem. In *Proc. of CP 2007*, volume 4741 of *LNCS*, pages 77–89. Springer, 2007.
17. Ian P. Gent, Tom Kelsey, Steve Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith. Conditional symmetry breaking. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming CP2005*, volume 3709 of *LNCS*, pages 256–270. Springer, 2005.
18. Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *Proceedings of the European Conference on Artificial Intelligence ECAI2000*, pages 599–603. IOS Press, 2000.
19. Lise Getoor, Greger Ottosson, Markus P. J. Fromherz, and Björn Carlson. Effective redundant constraints for online scheduling. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference*, pages 302–307, 1997.

20. Daniel Heller, Aurojit Panda, Meinolf Sellmann, and Justin Yip. Model restarts for structural symmetry breaking. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, pages 539–544. Springer, 2008.
21. Karla L. Hoffman and Manfred Padberg. Improving LP-representations of zero-one linear programs for branch-and-cut. *INFORMS Journal on Computing*, 3(2):121–134, 1991.
22. Toshihide Ibaraki. The power of dominance relations in branch-and-bound algorithms. *J. ACM*, 24(2):264–279, 1977.
23. J. Kalagnanam, M. Dawande, M. Trumbo, and H.S. Lee. Inventory matching problems in the steel industry. Technical report, IBM Research Report, T.J. Watson Research Center, 1998. RC 21171 (94615).
24. Richard E. Korf. Optimal rectangle packing: New results. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 142–149, 2004.
25. Roland Martin. The challenge of exploiting weak symmetries. In *Proc. of the International Workshop on Constraint Solving and Constraint Logic Programming*, volume 3978 of *LNCS*, pages 149–163. Springer, 2005.
26. Christopher Mears, Maria Garcia de la Banda, and Mark Wallace. On implementing symmetry detection. *Constraints*, 14(4):443–477, 2009.
27. H.E. Miller, W.P. Pierskalla, and G.J. Rath. Nurse scheduling using mathematical programming. *Operations Research*, pages 857–870, 1976.
28. J.N. Monette, P. Schaus, S. Zampelli, Y. Deville, and P. Dupont. A CP Approach to the Balanced Academic Curriculum Problem. In *Seventh International Workshop on Symmetry and Constraint Satisfaction Problems*, volume 7, 2007. `http://www.info.ucl.ac.be/~yde/Papers/SymCon2007_bacp.pdf`.
29. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming CP2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
30. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
31. Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming CP2004*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.
32. S. Prestwich and J.C. Beck. Exploiting dominance in three symmetric problems. In *Fourth International Workshop on Symmetry and Constraint Satisfaction Problems*, pages 63–70, 2004. `http://zeynep.web.cs.unibo.it/SymCon04/SymCon04.pdf`.
33. Les G. Proll and Barbara Smith. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS Journal on Computing*, 10(3):265–275, 1998.
34. PSPLib - project scheduling problem library. `http://129.187.106.231/psplib/`. Accessed on 1 March 2012.
35. Jean-Francois Puget. Automatic detection of variable and value symmetries. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming CP2005*, volume 3709 of *LNCS*, pages 475–489. Springer, 2005.
36. Andrea Rendl. *Effective compilation of constraint models.* PhD thesis, St Andrews University, 2010. http://hdl.handle.net/10023/973.
37. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1), 2008.
38. Chee Fen Yu and Benjamin W. Wah. Learning dominance relations in combinatorial search problems. *IEEE Trans. Software Eng.*, 14(8):1155–1175, 1988.