Dominance Driven Search

Geoffrey Chu[2] and Peter J. Stuckey[1,2]

[1] National ICT Australia, Victoria Laboratory
[2] Department of Computing and Information Systems,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au

Abstract. Recently, a generic method for identifying and exploiting dominance relations using *dominance breaking constraints* was proposed. In this method, sufficient conditions for a solution to be dominated are identified and these conditions are used to generate dominance breaking constraints which prune off the dominated solutions. We propose to use these dominance relations in a different way in order to boost the search for good/optimal solutions. In the new method, which we call *dominance jumping*, when search reaches a point where all solutions in the current domain are dominated, rather than simply backtrack as in the original dominance breaking method, we jump to the subtree which dominates the current subtree. This new strategy allows the solver to move from a bad subtree to a good one, significantly increasing the speed with which good solutions can be found. Experiments across a range of problems show that the method can be very effective when the original search strategy was not very good at finding good solutions.

1 Introduction

Recently, a generic method for identifying and exploiting dominance relations using *dominance breaking constraints* was proposed in [3]. This method analyzes the effects of different assignment mappings on the satisfiability and objective value of solutions, and finds sufficient conditions under which a solution is dominated by (i.e., no better than) another one. Dominance breaking constraints are then generated which prune off these dominated solutions.

While symmetry and dominance breaking constraints are very powerful and can produce orders of magnitude speedup on a wide range of problems, it is well known that static symmetry breaking constraints (e.g., [20, 4, 14, 21, 8]) can conflict with the search strategy, leading to less speedup or even a slowdown [10]. Static dominance breaking constraints, which are a generalization of static symmetry breaking constraints, suffer from a similar problem. In optimization problems, dominance breaking constraints prevent the solver from finding any solution which is dominated. While such dominated solutions may not be optimal, they may nevertheless improve the current best solution and allow additional pruning through branch and bound. If the search strategy quickly guides the search to good/optimal non-dominated solutions, then there is no conflict between the search and the dominance breaking constraints. However, if we have a search strategy which keeps pushing the search into subtrees with only dominated solutions, then the search will potentially conflict with the dominance breaking constraints.

Consider a situation where the next 100 solutions in the search tree are all dominated, but one or more of them does improve the current best solution. A search without dominance breaking constraints will go through that portion of the search space without the pruning provided by the dominance breaking constraints, but once one of these better solutions are found, it will gain some additional pruning from branch and bound. A search with dominance breaking constraints will have the pruning provided by the dominance breaking constraints, but none of the pruning from the better solution that it could have found among these 100 solutions. Thus it is possible that the solver is actually slower at finding the optimal solution with dominance breaking constraints than without.

In symmetry breaking, the potential conflict between search strategy and static symmetry breaking constraints can be overcome by using a dynamic symmetry breaking method such as symmetry breaking during search (SBDS) [1, 11] or symmetry breaking by dominance detection (SBDD) [5]. In this paper, we propose a different way to overcome this problem in the dominance case. We propose an altered method called *dominance jumping*, which exploits the information contained in the dominance relations in a different way. When a dominance breaking constraint prunes a partial assignment (because it is dominated by another), instead of simply backtracking as a normal CP solver would do, we jump to the subtree represented by the partial assignment which dominates the current one. Then, rather than getting stuck in a subtree where all the solutions are dominated and therefore not discoverable by a search with dominance breaking constraints, the dominance jumping can take us to a subtree with non-dominated solutions, allowing the search to find good non-dominated solutions and benefit from the additional pruning provided by branch and bound.

The rest of the paper is organized as follows: in the next section we introduce notation, and recall the approach to creating dominance breaking constraint described in [3]. In Section 3 we describe dominance jumping. In Section 4 we give experimental results comparing dominance breaking and dominance jumping. In Section 5 we examine related work, and finally in Section 6 we conclude.

2 Definitions

2.1 Constraints, Literals, and COPs

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 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., $\mathbb{Z}$ or $\mathbb{R}$, 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.

CP solvers solve CSP's by interleaving search with inference. We begin with the original problem at the root of the search tree. At each node in the search tree, we propagate the constraints to try to infer variable/value pairs which can no longer be taken in any solution in this subtree. Such pairs are removed from the current domain. If some variable's domain becomes empty, then the subtree 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, the solver further divides the problem into a number of more constrained subproblems and searches each of those in turn. COP's are typically solved via branch and bound where we solve a series of CSP's with increasingly tight bounds on the objective value.

2.2 Dominance breaking

Without loss of generality, assume that we are dealing with a minimization problem. We say that assignment θ_1 dominates θ_2 if either: 1) θ_1 is a solution and θ_2 is a non-solution, or 2) they are both solutions or both non-solutions and $f(\theta_1) \leq f(\theta_2)$. In [3], a generic method for identifying and exploiting dominance relations via dominance breaking constraints was proposed. The method can be briefly outlined as follows:

Step 1 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 2 Find mappings $\sigma : \Theta_V \rightarrow \Theta_V$ which are likely to map solutions to better solutions.
Step 3 For each σ, find a constraint $scond(\sigma)$ s.t. if $\theta \in solns(C \wedge D \wedge scond(\sigma))$, then $\sigma(\theta) \in solns(C \wedge D)$.
Step 4 For each σ, find a constraint $ocond(\sigma)$ s.t. if $\theta \in solns(C \wedge D \wedge ocond(\sigma))$, then $f'(\sigma(\theta)) < f'(\theta)$.
Step 5 For each σ, post the dominance breaking constraint $db(\sigma) \equiv \neg(scond(\sigma) \wedge ocond(\sigma))$.

The method analyzes the effects of different assignment mappings σ on the satisfiability and objective value of solutions, and finds sufficient conditions $scond(\sigma) \wedge ocond(\sigma)$ under which a solution is dominated by another one. Dominance breaking constraints $db(\sigma) \equiv \neg(scond(\sigma) \wedge ocond(\sigma))$ are then generated which prune off these dominated solutions. We now restate the main theorem from [3] showing the correctness of the dominance breaking constraints generated by this method.

Theorem 1. *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 $scond(\sigma)$ and $ocond(\sigma)$ satisfying: $\forall \sigma \in S$,* *if $\theta \in solns(C \wedge D \wedge scond(\sigma))$, then $\sigma(\theta) \in solns(C \wedge D)$, and: $\forall \sigma \in S$, if $\theta \in solns(C \wedge D \wedge ocond(\sigma))$, then $f'(\sigma(\theta)) < f'(\theta)$, we can add all of the dominance breaking constraints $db(\sigma) \equiv \neg(scond(\sigma) \wedge ocond(\sigma))$ to P without changing its satisfiability or optimal value.*

A proof of this theorem and more details on the method can be found in [3].

Example 1. Consider the 0-1 knapsack problem where x_i are 0-1 variables, we have constraint $\sum w_i x_i \leq W$ and we have objective $f = -\sum v_i x_i$, where w_i and v_i are constants.

Step 1 Initially, let us not refine the objective function leaving $f' = f$.

Step 2 Consider mappings which swap the values of two variables, i.e., $\forall i < j, \sigma_{i,j}$ swaps x_i and x_j.

Step 3 A sufficient condition for $\sigma_{i,j}$ to map the current solution to another solution is: $scond(\sigma_{i,j}) \equiv w_i x_j + w_j x_i \leq w_i x_i + w_j x_j$. Rearranging, we get: $(w_i - w_j)(x_i - x_j) \geq 0$.

Step 4 A sufficient condition for $\sigma_{i,j}$ to map the current solution to an assignment with a better objective value is: $ocond(\sigma_{i,j}) \equiv v_i x_j + v_j x_i > v_i x_i + v_j x_j$. Rearranging, we get: $(v_i - v_j)(x_i - x_j) < 0$.

Step 5 For each $\sigma_{i,j}$, we can post the dominance breaking constraint: $db(\sigma_{i,j}) \equiv \neg(scond(\sigma_{i,j}) \wedge ocond(\sigma_{i,j}))$. After simplifying, we have $db(\sigma_{i,j}) \equiv x_i \leq x_j$ if $w_i \geq w_j$ and $v_i < v_j$, $db(\sigma_{i,j}) \equiv x_i \geq x_j$ if $w_i \leq w_j$ and $v_i > v_j$, and $db(\sigma_{i,j}) \equiv true$ for all other cases.

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

We can refine the objective to get stronger dominance breaking constraints. In Step 1, we can tie break solutions with equal objective value by the weight used, and then lexicographically, i.e., $f' = lex(f, \sum w_i x_i, x_1, \ldots, x_n)$. In Step 4, we have: $\forall i < j, ocond(\sigma_{i,j}) \equiv \sigma(f') < f' \equiv ((v_i - v_j)(x_i - x_j) < 0) \vee ((v_i - v_j)(x_i - x_j) = 0 \wedge (w_i - w_j)(x_i - x_j) > 0) \vee ((v_i - v_j)(x_i - x_j) = 0 \wedge (w_i - w_j)(x_i - x_j) = 0 \wedge x_j < x_i)$. In Step 5, after simplifying, in addition to the dominance breaking constraints we had before, we would also have: $db(\sigma_{i,j}) \equiv x_i \leq x_j$ if $w_i > w_j$ and $v_i = v_j$, $db(\sigma_{i,j}) \equiv x_i \geq x_j$ if $w_i < w_j$ and $v_i = v_j$, and $db(\sigma_{i,j}) \equiv x_i \leq x_j$ if $w_i = w_j$ and $v_i = v_j$ which is a symmetry breaking constraint. □

3 Dominance Jumping

A propagator for a dominance breaking constraint can do two things: 1) it can check the consistency of the current domain w.r.t. the dominance breaking constraint, producing failure if it is inconsistent, and 2) it can prune off variable/value pairs which, if taken, will cause inconsistency. In the original method, the failure and propagation produced by these propagators are treated the same as any other propagator in the system. In our altered method, whenever dominance jumping is active, we modify this behavior as follows: 1) we check consistency only and never prune any values using the dominance breaking constraints, 2) when a failure is detected, we perform a *dominance jump* rather than a normal backtrack.

As can be seen from the definitions in Section 2, each dominance breaking constraint $db(\sigma)$ is generated from an assignment mapping σ. When a domain D is failed by $db(\sigma)$, it means that every solution θ in D is dominated by a corresponding solution $\sigma(\theta)$. Rather than simply failing and backtracking, we can instead perform a *dominance jump* to get to the part of the search tree that contains these better solutions. Let us extend σ to also map domains to domains via $solns(\sigma(D)) \equiv \{\sigma(\theta) \mid \theta \in solns(D)\}$. Then, if D is failed by $db(\sigma)$, the domain $\sigma(D)$ must contain solutions which dominate those in the current domain D. We want to calculate this new domain $\sigma(D)$ and jump to there. In this paper, we consider only σ which are literal mappings, i.e., assignment mappings which map each equality literal $x = v$ to the same or another equality literal $x' = v'$ in all assignments. All of the σ used in [3] are literal mappings, and indeed we expect that most practically useful mappings for the method proposed in [3] will be literal mappings. Let us extend σ to map equality literals to equality literals and disequality literals to disequality literals such that if $\sigma(x = v) = (x' = v')$, then $\sigma(x \neq v) = (x' \neq v')$.

Any domain D can be expressed as a set of disequality literals $lits_D$ representing which variable/value pairs from the original domain has been pruned. For example, suppose the initial domain D_{init} of x_1, x_2 were $\{1,2,3,4,5\}$ and the current domain D is $x_1 \in \{1,3,5\}, x_2 \in \{2,3,4\}$. Then $lits_D \equiv \{x_1 \neq 2, x_1 \neq 4, x_2 \neq 1, x_2 \neq 5\}$. Using the set of literals $\sigma(lits_D) \equiv \{\sigma(l) \mid l \in lits_D\}$ as decisions from the root node will take us to the search space $\sigma(D)$. We do this by backtracking to the deepest level such that all previous decisions in the current search path are in $\sigma(lits_D)$. We then suspend the normal search strategy and draw decisions from $\sigma(lits_D)$ until it is either exhausted, or some conflict occurs. After that, we resume the normal search strategy. If D had variables which were fixed, we can use those equality literals in $lits_D$ instead so we need to make fewer decisions to get to $\sigma(D)$. Similarly, if σ happens to also map inequality literals to inequality literals (e.g., in a mapping which swaps variables), we can use those to reduce the number of decisions.

Example 2. 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 represent 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: $alldiff([x_1,\ldots,x_n])$. The objective function to be maximised is given by: $f = \sum_{i=1}^{n-1} p[x_i][x_{i+1}]$. In MiniZinc [18] it would be modelled as:

```
int: n;                            % number of people
set of int: Person = 1..n;
array[Person,Person] of 0..2: p;   % preferences

array[Person] of Person: x;        % person in position i;

constraint alldifferent(x);
```

```
solve maximize sum(i in 1..n-1)(p[x[i],x[i+1]]);
```

As described in [3], if we consider the mappings $\sigma_{i,j}$ which flip a subsequence of the x's from the ith position to the jth (i.e., map x_i to x_j, x_{i+1} to x_{j-1}, $\ldots$, x_j to x_i), we can generate a number of dominance breaking constraints. For $2 \geq i < j \leq n-1$, $db(\sigma_{i,j}) \equiv p[x_{i-1}][x_i] + p[x_j][x_{j+1}] + (x_i < x_j) > p[x_{i-1}][x_j] + p[x_i][x_{j+1}]$. For $i = 1, 2 \leq j$, $db(\sigma_{i,j}) \equiv p[x_j][x_{j+1}] + (x_i < x_j) > p[x_i][x_{j+1}]$. For $i \leq n-1, j = n$, $db(\sigma_{i,j}) \equiv p[x_{i-1}][x_i] + (x_i < x_j) > p[x_{i-1}][x_j]$. For $i = 1, j = n$, $db(\sigma_{i,j}) \equiv (x_i < x_j) > 0$.

We now illustrate the difference between dominance breaking and dominance jumping on a simple example. Suppose $n = 6$ and person 1 wants to stand next to person 6, person 6 wants to stand next to person 2, person 2 wants to stand next to person 5, person 5 wants to stand next to person 3, and person 3 wants to stand next to person 4. This is expressed by the MiniZinc data file:

```
n = 6;
p = [| 0, 0, 0, 0, 0, 1
     | 0, 0, 0, 0, 1, 1
     | 0, 0, 0, 1, 1, 0
     | 0, 0, 1, 0, 0, 0
     | 0, 1, 1, 0, 0, 0
     | 1, 1, 0, 0, 0, 0 |];
```

Suppose we use a naive search strategy such as labelling the x_i in order, trying the lowest value available in the domain first. With neither dominance breaking nor dominance jumping, it takes the search 51 conflicts to reach an optimal solution $x_1 = 1, x_2 = 6, x_3 = 2, x_4 = 5, x_5 = 3, x_6 = 4$. With dominance breaking, the search proceeds as follows. At the first decision level, we try $x_1 = 1$. At the second decision level, we try $x_2 = 2$. At this point, the constraint $p[x_1][x_2] + (x_2 < x_6) > p[x_1][x_6]$ propagates to force $p[x_1][x_6] = 0$, which forces $x_6 \neq 6$. At the third decision level, we try $x_3 = 3$. At this point, the constraint $p[x_2][x_3] + (x_3 < x_6) > p[x_2][x_6]$ propagates to force $p[x_2][x_6] = 0$, which forces $x_6 \neq 5$, which forces $x_6 = 4$. The constraint $p[x_3][x_4] + (x_4 < x_6) > p[x_3][x_6]$ now propagates and forces $p[x_3][x_4] \geq 2$ because $p[x_3][x_6] = p[3][4] = 1$, and x_4 is either 5 or 6 so $(x_4 < x_6) = 0$. But then $x_4 \neq 5$ and $x_4 \neq 6$ and we have a failure. We then backtrack and continue the search. After another 25 conflicts, we reach the optimal solution. The search tree is shown in Figure 1.

With dominance jumping, the search proceeds as follows. We make the 5 decisions $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5$, which forces $x_6 = 6$. At this point, a number of dominance breaking constraints are detected to be violated, telling us that certain mappings can improve the solution. For example, $db(\sigma_{2,6})$ is violated. Using the mapping $\sigma_{2,6}$ to perform a jump means that we backtrack to decision level 1, and then try the decisions $x_2 = 6, x_3 = 5, x_4 = 4, x_5 = 3, x_6 = 2$. After these, we again detect that a dominance breaking constraint is violated, e.g., $db(\sigma_{3,6})$. Performing this jump means that we backtrack to decision level 2 and try $x_3 = 2, x_4 = 3, x_4 = 4, x_6 = 5$. This violates $db(\sigma_{4,6})$. Applying this jump causes us to backtrack to decision level 3 and try $x_4 = 5, x_5 = 4, x_6 = 3$. This violates $db(\sigma_{5,6})$. Applying this jump causes us to backtrack to decision level 4 and try $x_5 = 3, x_6 = 4$, finally giving a non-dominated solution of $x_1 = 1, x_2 = 6, x_3 = 2, x_4 = 5, x_5 = 3, x_6 = 4$. In this case, it only took 4 conflicts and 4 jumps to bring us to the optimal solution. The search tree is shown in Figure 2. □

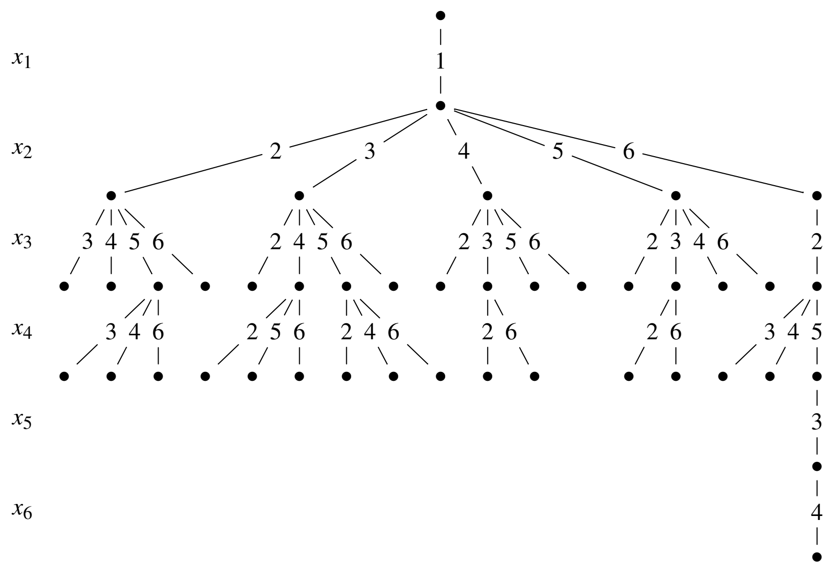

Fig. 1. Search tree with dominance breaking.

The effect of dominance jumping is significantly different from dominance breaking. When a "bad" decision is made (e.g., $x_2 = 2$ after $x_1 = 1$ in Example 2), dominance breaking is incapable of "fixing" the problem. Instead, it just helps the solver to fail that bad subtree quicker so that it can backtrack out of the bad decision. However, in general, it still takes exponential time to undo the bad decision. Dominance jumping on the other hand, can potentially fix a bad decision and replace it with a good one very quickly. In Example 2, after $x_1 = 1, x_2 = 2$, if some x_i is set to 6, it is likely that flipping the subsequence from 2 to i improves the objective. Thus dominance jumping will jump to a subtree where $x_1 = 1, x_2 = 6$, immediately undoing the bad decision $x_2 = 2$.

Note that dominance jumping does not require all variables involved in the dominance breaking constraint to be fixed in order to jump.

Example 3. Consider a simple problem: $x_1 + x_2 + x_3 + x_4 \leq 9 \wedge \mathit{alldiff}(x_1, x_2, x_3, x_4)$ with $D(x_i) = [1..6]$. All variables are symmetric. So $\sigma_{i,j}$ which swaps the values of x_i and x_j is a mapping that preserves solutions. Using a lexicographic objective f' we can compute $db(\sigma_{i,j}) = x_i \leq x_j$ for $i < j$. Imagine we label $x_2 = 1$, then propagation causes $D(x_1) = [2..6]$ and $db(\sigma_{1,2})$ fails. We compute $\sigma_{1,2}(D)$ as $D(x_2) = [2..6]$ and $D(x_1) = \{1\}$. The dominance jump goes to the root and then sets $x_1 = 1$ (which will set $D(x_2) = [2..6]$) and then continues its search. Of course for this trivial example dominance breaking is clearly superior. □

Both dominance breaking and dominance jumping are optimality preserving. As Theorem 1 states, the addition of dominance breaking constraints to the problem does not change its satisfiability or optimal value. Performing dominance jumping will not

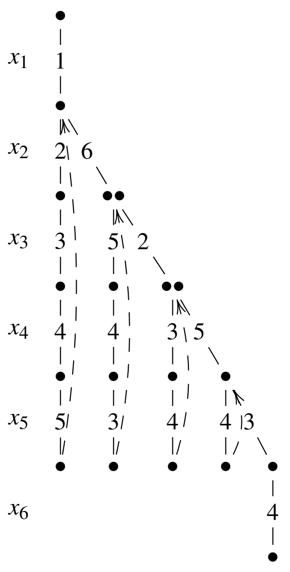

Fig. 2. Search tree with dominance jumping.

change its satisfiability or optimal value either. When a dominance jump is performed, nothing is actually pruned. We are only doing a restart and temporarily using a different search strategy to guide the solver to another part of the search space. The subtree we jumped away from is not counted as fully searched, so dominance jumping can never prune off optimal solutions of the problem. However, there can be issues of terminability when dominance jumping is used, as if we keep jumping, we may never complete a full search of the search tree. Nogood learning techniques such as [13, 19, 6]) can be used to overcome this problem. Such techniques record nogoods which tell us which parts of the search tree the solver has proved contains no solution better than the current best solution. Such nogoods allow the solver to keep track of which subtrees have been exhaustively searched. If we keep all such nogoods permanently, the search will always terminate and will be complete, guaranteeing that the correct optimal value is found. We implemented dominance jumping in the nogood learning solver CHUFFED which already has nogood learning built in, and thus it is capable of performing a complete search with dominance jumping.

The effectiveness of dominance jumping depends significantly on how good the original search strategy was. If the original search strategy was already good enough that it very quickly leads the search to an optimal solution, then dominance jumping is largely pointless. On the other hand, if the original search strategy was quite naive, or is not designed for finding good solutions, then dominance jumping can be very useful. It is quite common that the search strategies used in CP solvers are not good at finding good solutions. This is because many of them are designed to reduce the size of the search tree, rather than to order the subtrees so that good solutions are found first. Many of the common dynamic search strategies such as first fail [12], dom/wdeg and its variants [2], and activity based search [17] are purely designed to make the search tree smaller and makes no effort whatsoever to try to guide the solver to good solutions quickly. But when dominance jumping is used on top of them, even very naive search strategies which are bad at finding solutions can turn into good ones, as a significant amount of information about the objective and the structure of the problem is contained in the dominance relations. By exploiting this through dominance jumping, we can bring the solver to a good subtree even if the search strategy initially sent it to a bad one.

While dominance jumping can be effective in the solution finding phase of an optimization problem, it is completely useless in the proof of optimization phase. In that phase, we are no longer trying to find any solutions. Jumping around in the search space will simply make it more difficult to complete the proof of optimality. Nogood learning can prevent thrashing behavior caused by the jumping and allow a complete proof of optimality to be derived. However, for large/hard problems, this may require an unreasonably large number of nogoods to be stored, causing the solver to run out of memory. Ideally, if a proof of optimality is desired, we should use some dynamic method to switch off dominance jumping and go back to pure dominance breaking. Such strategies will be explored in future work.

4 Experiments

The experiments were performed on Xeon Pro 2.4GHz processors using the state of the art CP solver CHUFFED. We use a time out of 600 seconds. We compare the base solver vs dominance breaking and dominance jumping. We use the 7 optimization problems used in [3]: `Photo-`*n* the photo problem of Example 2 with *n* people, `Knapsack-`*n* 0-1 knapsack problem of Example 1 with *n* objects, `Nurse-`*n* nurse scheduling problem [15] with 15 nurses and *n* days, `RCPSP` resource constrained scheduling problem J120 instances, `Talent-`*n* talent scheduling problem [9] with *n* scenes, `SteelMill-`*n* steel mill scheduling [10] with *n* orders, and `PCBoard-`*n*-*m* PB board manufacturing problem with *n* components and *m* devices. We use basic inorder fixed search strategies that are not specifically designed to find good solutions. MiniZinc models and data for the problems can be found at: `www.cs.mu.oz.au/~pjs/dom-jump/`

Results are shown in Table 1: *opt* is average of the best solution found; *otime* is the geometric mean of time to find the best solution; *etime* is the geometric mean of time to find a solution at least as good as the worser of the best solution found by dominance breaking and the best solution found by dominance jumping, so we can directly compare *etime* to see how much time each took to get the same quality solution; *stime* is the geometric mean of the time to solve the instance completely (timeouts count as 600); and finally *svd* the number of instances solved to optimality is given in brackets. The best values out of the three methods are given in bold. When there is a tie on the best value, we tie-break on the time required to achieve the value.

The results show that both dominance breaking and dominance jumping substantially improve upon solving without dominance information. Dominance jumping is clearly better at finding good solutions faster. The average best solution found is almost always better. Dominance jumping almost always wins in *etime*, the only exception is in smaller knapsacks and in nurse scheduling where dominance breaking is obviously far superior. Notice how, as the difficulty of the instances grows with size, dominance jumping becomes more advantageous.

Table 1. Comparison of the solver with nothing (none), with dominance breaking constraints (dominance breaking), and with dominance jumping. dominance jumping

Problem	none				dominance breaking					dominance jumping				
	opt	*otime*	*stime*	*svd*	*opt*	*otime*	*etime*	*stime*	*svd*	*opt*	*otime*	*etime*	*stime*	*svd*
Photo-16	19.5	16.43	50.13	18	19.7	1.03	1.03	**11.40**	**20**	**19.7**	0.10	**0.10**	16.64	20
Photo-18	21.4	39.45	182.7	15	21.5	2.27	2.27	**76.26**	**20**	**21.5**	0.29	**0.29**	80.62	19
Photo-20	21.4	179.6	393.3	5	23.15	31.06	31.06	262.4	7	**23.15**	1.16	**1.16**	**232.9**	**9**
Photo-22	22.8	185.1	368.7	4	25.15	51.56	38.9	294.7	6	**25.4**	2.18	**0.76**	**244.4**	**7**
Photo-24	21.55	213.5	596.5	1	26.75	147.9	147.9	586.7	3	**27.15**	2.00	**0.52**	**495.6**	**4**
Knapsack-100	1827.95	222.3	600	0	**2583.2**	0.30	**0.30**	**0.93**	**20**	2583.2	0.60	0.60	1.52	20
Knapsack-150	3605.75	240.6	600	0	**5810.35**	13.8	**13.82**	**47.62**	**19**	5810.35	27.71	27.71	82.26	19
Knapsack-200	5910.55	299.2	600	0	**10422.4**	180.1	**126.2**	**261.0**	**3**	10415.8	220.7	206.2	491.9	2
Knapsack-250	8732.25	342.4	600	0	16235.25	253.9	186.2	600	0	**16235.5**	327.1	**151.7**	600	0
Knapsack-300	12000	288.9	600	0	23212	272.6	157.2	600	0	**23294.9**	302.2	**123.0**	600	0
Nurse-14	149.2	61.35	61.82	18	**151.35**	50.99	**49.14**	**52.33**	**19**	150.2	72.45	72.45	74.06	18
Nurse-21	137.1	416.5	600	0	**172.7**	248.0	**217.6**	600	0	172.4	280.7	254.3	600	0
Nurse-28	161.5	400.5	600	0	222.4	245.1	**213.5**	600	0	**222.45**	310.2	277.6	600	0
Nurse-35	187.5	355.0	600	0	**275.75**	339.8	**164.7**	600	0	275.1	223.7	196.4	600	0
Nurse-42	213.1	382.3	600	0	**321.9**	377.4	**193.3**	600	0	320.55	332.9	332.9	600	0
RCPSP	110.9	31.37	41.62	72	114.17	7.51	7.51	13.97	57	**110.86**	12.44	**3.92**	**15.83**	**72**
Talent-16	106.1	7.58	19.03	20	106.05	0.92	0.92	3.25	20	**106.05**	0.41	**0.41**	**2.79**	**20**
Talent-18	149.45	127.1	239.3	16	147	5.22	5.22	17.26	20	**147**	2.73	**2.73**	**15.81**	**20**
Talent-20	270.2	321.4	497.3	5	184.45	27.22	27.22	81.48	20	**184.45**	14.85	**14.85**	**73.40**	**20**
Talent-22	387.1	369.2	600	0	270.9	34.32	34.32	353.2	13	**204.05**	51.4	**21.10**	**310.5**	**14**
Talent-24	566.65	323.8	600	0	322.25	161.6	154.9	547.1	2	**260.1**	123.4	**18.27**	**510.7**	**3**
SteelMill-40	5.45	71.02	129.3	10	1.6	42.30	40.80	54.60	15	**0.65**	20.24	**9.33**	**21.75**	**17**
SteelMill-45	8.2	105.7	269.6	6	1.55	121.7	121.7	134.1	14	**0.35**	46.44	**31.92**	**52.83**	**18**
SteelMill-50	16.35	319.6	560.1	1	9.85	91.00	90.73	332.1	10	**1.1**	142.1	**30.75**	**198.9**	**16**
SteelMill-55	25.95	305.0	600	0	16.05	67.26	60.19	419.3	6	**2.9**	212.2	**23.59**	**275.6**	**13**
SteelMill-60	32.55	274.4	584.8	1	19.9	59.56	54.65	497.0	2	**5**	224.7	**23.82**	**399.0**	**8**
PCBoard-6-8	206.25	298.8	341.1	8	217.7	17.50	17.50	**24.36**	**20**	**217.7**	1.29	**1.29**	68.19	14
PCBoard-6-9	225.9	440.2	532.8	2	**246.7**	84.81	80.04	**142.5**	**20**	246.6	8.39	**8.39**	438.5	3
PCBoard-7-9	242.15	418.6	600	0	277.7	123.5	111.4	**498.5**	**7**	**282.5**	21.95	**5.56**	600	0
PCBoard-7-10	266.9	383.7	600	0	282.55	6.32	6.32	600	0	**319.15**	28.48	**0.86**	600	0
PCBoard-8-10	293.9	379.2	600	0	308.4	4.35	4.35	**597**	**1**	**357.9**	30.32	**0.38**	600	0

Unsuprisingly dominance jumping is usually better at proving optimality having better *stime* in most of the smaller instances. Suprisingly dominance jumping actually turns out to be preferable to dominance breaking even for proving optimality on RCPSP, Talent and SteelMill. For these problems the proof of optimality is not the larger part of the search space, that is once we find the optimal solution for these problems it is often no too difficult to prove it optimal.

Next we compare the three methods using 3 different search heuristics for the Photo problem, to see how the search strategy affects the effectiveness of dominance breaking and dominance jumping. The first is the basic inorder fixed search used above (inorder). The second greedily finds a person who most wants to stand next to an already assigned person and assigns them next to them (greedy). The third uses the first fail heuristic to pick which variable to assign next (first-fail).

Table 2. Comparison of dominance breaking and dominance jumping given different search heuristics.

Problem	Search	none				dominance breaking					dominance jumping				
		opt	*otime*	*stime*	*svd*	*opt*	*otime*	*etime*	*stime*	*svd*	*opt*	*otime*	*etime*	*stime*	*svd*
Photo-16	inorder	19.65	7.37	11.47	18	19.70	0.79	0.79	2.35	20	**19.70**	0.08	**0.08**	**0.98**	**20**
	greedy	19.70	1.29	3.44	19	19.70	0.45	0.45	1.71	20	**19.70**	0.04	**0.04**	**0.56**	**20**
	first-fail	19.70	0.72	1.22	20	19.70	0.24	0.24	**0.65**	**20**	**19.70**	0.13	**0.13**	0.76	20
Photo-18	inorder	21.45	18.47	45.25	19	21.50	1.42	1.42	9.71	20	**21.50**	0.20	**0.20**	**5.60**	**20**
	greedy	21.50	0.69	7.80	19	21.50	0.39	0.39	4.09	20	**21.50**	0.08	**0.08**	**2.72**	**20**
	first-fail	21.50	1.05	2.79	20	21.50	0.28	0.28	**0.97**	**20**	**21.50**	0.24	**0.24**	1.62	20
Photo-20	inorder	22.50	161.35	228.98	10	23.20	16.74	16.74	54.97	16	**23.20**	1.23	**1.23**	**15.92**	**16**
	greedy	23.00	5.36	38.38	12	23.20	3.26	3.26	19.01	18	**23.20**	0.10	**0.10**	**3.29**	**18**
	first-fail	23.20	4.90	12.95	17	23.20	1.17	1.17	**3.45**	**20**	**23.20**	0.54	**0.54**	3.46	20
Photo-22	inorder	23.70	137.69	254.99	55	25.35	44.46	44.46	102.18	13	**25.40**	2.09	**1.57**	**12.27**	**14**
	greedy	25.15	8.28	60.94	11	25.40	9.54	9.54	36.68	15	**25.45**	0.60	**0.60**	**5.38**	**15**
	first-fail	25.45	13.94	19.71	16	25.50	3.36	3.36	**6.42**	**20**	**25.50**	2.33	**2.33**	6.55	20
Photo-24	inorder	22.45	216.79	519.93	3	26.85	87.71	87.71	350.3	7	**27.10**	1.36	**0.71**	**61.88**	**8**
	greedy	26.60	11.75	184.26	7	26.95	16.52	16.52	171.90	8	**27.35**	0.66	**0.23**	**27.71**	**12**
	first-fail	26.55	43.17	88.59	12	27.45	13.10	13.10	33.29	19	**27.45**	5.12	**5.12**	**20.97**	**19**

The results in Table 2 show that dominance jumping is still much better at reaching a good solution than dominance breaking, regardless of the search strategy. The results clearly illustrate that the biggest advantage of dominance jumping arises in the inorder search, which does not try to look for good solutions. But dominance jumping is still advantageous over dominance breaking using the greedy search, although to a lesser degree. Using first-fail dominance breaking is better at proving optimality, since first-fail search also concentrates on reducing the search space, but as the size of the problem grows, its advantage reduces, until for Photo-24 dominance jumping is superior in proving optimality as well.

5 Related Work

Dominance breaking constraints were introduced only recently in [3] and there has been little work analyzing how they may conflict with the search or how that problem can be overcome. The closest related work is that for the special case of symmetry breaking. In this case, potential conflicts between the search and static symmetry breaking constraints can be overcome by using dynamic symmetry breaking techniques such as SBDS [1, 11] or SBDD [5]. However, neither of these methods obviously generalize to the dominance case. In the case of symmetry, we have sets of equally good symmetric subtrees. The policy in SBDS/SBDD is to search the first member of each such set encountered during search, and to prune all other members as soon as they are encountered. In dominance breaking however, we have pairs of subtrees where one may be strictly better than the other (i.e., contains a strictly better solution). The ordering between them is not up to us to decide as it is determined by the search strategy, and we cannot simply decide to always search the first of the pair and prune the second. We could try a different policy such as: if we encounter the good one first, we prune the bad one later, and if we encounter the bad one first, we search both. Indeed, such a policy was proposed in [7]. However, checking whether a subtree is dominated by any of the previously searched subtrees is extremely complex in general, and is much harder than simply determining whether it is dominated by some (possibly not yet explored) subtree. In [7], it is shown how an incomplete version of such a dominance check can be performed using greedy local search for the Travelling Salesman Problem. However, it is not clear how complete it is or whether it can generalise to other problems. Also, even if the dominance check can be performed efficiently, such a method will still perform more search in general than the dominance jumping method presented here. In dominance jumping, regardless of the order in which we encounter the pair of subtrees, we will only search the good one and will always skip the bad one, because if we encounter the bad one first, we will simply immediately jump to the good one.

Dominance jumping shares some features with local search/repair methods such as min-conflict search [16]. However, such methods typically travel through the space of infeasible solutions and jump at every node. Dominance jumping on the other hand is a systematic search which remains within feasible space, and only occasionally jumps when we are guaranteed to get to a better subtree.

Dominance jumping is also related to best first search. When best first search reaches a node which is recognized as possibly dominated (since the lower bound on the objective is substantially worse than another part of the search tree), it jumps to what it thinks is the best node and explores from there. In this case the jump is a heuristic, unlike in dominance jumping where we have a proof that the current node is suboptimal and we jump to a strictly better node.

6 Conclusion

We have developed a new method called dominance jumping to exploit the dominance relations identified by the method proposed in [3]. Rather than failing and backtracking as in the original method, we use the dominance relation to jump to a different part of the search tree that dominates the current subtree. Unlike static dominance breaking constraints, the new method will not conflict with the search strategy. Experimental evidence shows that the method allows good/optimal solutions to be found much more quickly on a wide range of problems. Important future work is to examine how to automatically determine when during search to switch from dominance jumping to dominance breaking, so that we can take advantage of the strengths of both approaches.

Acknowledgments NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In *Proc. CP 1999*, volume 1713 of *LNCS*, pages 73–87. Springer, 1999.

2. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Procs. of ECAI04*, pages 146–150, 2004.
3. G. Chu and P.J. Stuckey. A generic method for systematically identifying and exploiting dominance relations. In *Proc. CP 2012*, number 7514 in LNCS, pages 6–22. Springer, 2012.
4. 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.
5. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference*, pages 93–107, 2001.
6. Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *Proc. of CP 2009*, volume 5732 of *LNCS*, pages 352–366. Springer, 2009.
7. Filippo Focacci and Paul Shaw. Pruning sub-optimal search branches using local search. In *CPAIOR*, volume 2, pages 181–189, 2002.
8. Alan M. Frisch, Christopher Jefferson, and Ian Miguel. Constraints for Breaking More Row and Column Symmetries. In Francesca Rossi, editor, *Proc. of CP 2003*, volume 2833 of *LNCS*, pages 318–332. Springer, 2003.
9. 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.
10. 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.
11. I. Gent and B.M. Smith. Symmetry breaking in constraint programming. In *14th European Conference on Artificial Intelligence*, pages 599–603, 2000.
12. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 1980.
13. Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Nogood Recording from Restarts. In Manuela M. Veloso, editor, *Proc. of IJCAI 2007*, pages 131–136, 2007.
14. Eugene M. Luks and Amitabha Roy. The Complexity of Symmetry-Breaking Formulas. *Ann. Math. Artif. Intell.*, 41(1):19–45, 2004.
15. H.E. Miller, W.P. Pierskalla, and G.J. Rath. Nurse scheduling using mathematical programming. *Operations Research*, pages 857–870, 1976.
16. Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1):161–205, 1992.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Procs. of DAC2001*, pages 530–535, 2001.
18. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *Proc. of CP 2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
19. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In *Proc. of CP 2007*, volume 4741 of *LNCS*, pages 544–558. Springer, 2007.
20. Jean Francois Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In Henryk Jan Komorowski and Zbigniew W. Ras, editors, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, volume 689 of *LNCS*, pages 350–361. Springer, 1993.
21. Jean Francois Puget. Breaking symmetries in all different problems. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proc. of IJCAI 2005*, pages 272–277. Professional Book Center, 2005.