Automatically Exploiting Subproblem Equivalence in Constraint Programming

Geoffrey Chu[1], Maria Garcia de la Banda[2], and Peter J. Stuckey[1]

[1] National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au
[2] Faculty of Information Technology,
Monash University, Australia
mbanda@infotech.monash.edu.au

Abstract. Many search problems contain large amounts of redundancy in the search. In this paper we examine how to automatically exploit remaining subproblem equivalence, which arises when two different search paths lead to identical remaining subproblems, that is the problem left on the remaining unfixed variables. Subproblem equivalence is exploited by caching descriptions, or keys, that define the subproblems visited, and failing the search when the key for the current subproblem already exists in the cache. In this paper we show how to automatically and efficiently define keys for arbitrary constraint problems. We show how a constraint programming solver with this capability can solve search problems where subproblem equivalence arises orders of magnitude faster. The system is fully automatic, i.e., the subproblem equivalences are detected and exploited without any effort from the problem modeller.

1 Introduction

When solving a search problem, it is common for the search to do redundant work, due to different search paths leading to subproblems that are somehow "equivalent". There are a number of different methods to avoid this redundancy, such as caching solutions (e.g. [19]), symmetry breaking (e.g. [8]), and nogood learning (e.g. [14]). This paper focuses on caching, which works by storing information in a cache regarding every new subproblem explored during the search. Whenever a new subproblem is about to be explored, the search checks whether there is an already explored subproblem in the cache whose information (such as solutions or a bound on the objective function) can be used for the current subproblem. If so, it does not explore the subproblem and, instead, uses the stored information. Otherwise, it continues exploring the subproblem. For caching to be efficient, the lookup operation must be efficient. A popular way is to store the information using a *key* in such a way that problems that can reuse each other's information are mapped to the same (or similar) key.

This paper explores how to use caching *automatically* to avoid redundancy in constraint programming (CP) search. Caching has been previously used in CP

A. Lodi, M. Milano, and P. Toth (Eds.): CPAIOR 2010, LNCS 6140, pp. 71–86, 2010.
© Springer-Verlag Berlin Heidelberg 2010

search, but either relies on the careful manual construction of the key for each model and search strategy (e.g [19]), or exploits redundancy when the remaining subproblem can be decomposed into independent components (e.g. [10,12]). Instead, we describe an approach that can automatically detect and exploit caching opportunities in arbitrary optimization problems, and does not rely on decomposition. The principal insight of our work is to define a key that can be efficiently computed during the search and can uniquely identify a relatively general notion of reusability (called U-dominance). The key calculation only requires each primitive constraint to be extended to *backproject* itself on the fixed variables involved. We experimentally demonstrate the effectiveness of our approach, which has been implemented in a competitive CP solver, CHUFFED. We also provide interesting insight into the relationships between U-dominance and dynamic programming, symmetry breaking and nogood learning.

2 Background

Let $\equiv$ denote syntactic identity and $vars(O)$ denote the set of variables of object O. A *constraint problem* P is a tuple (C, D), where D is a set of *domain constraints* of the form $x \in s_x$ (we will use $x = d$ as shorthand for $x \in \{d\}$), indicating that variable x can only take values in the fixed set s_x, and C is a set of constraints such that $vars(C) \subseteq vars(D)$. We will assume that for every two $x \in s_x, y \in s_y$ in $D : x \not\equiv y$. We will define D_V, the restriction of D to variables V, as $\{(x \in s_x) \in D | x \in V\}$. Each set D and C is logically interpreted as the conjunction of its elements.

A *literal* of $P \equiv (C, D)$ is of the form $x \mapsto d$, where $\exists (x \in s_x) \in D$ s.t. $d \in s_x$. A *valuation θ of P over set of variables $V \subseteq vars(D)$* is a set of literals of P with exactly one literal per variable in V. It is a mapping of variables to values. The *projection* of valuation θ over a set of variables $U \subseteq vars(\theta)$ is the valuation $\theta_U = \{x \mapsto \theta(x) | x \in U\}$. We denote by $fixed(D)$ the set of fixed variables in D, $\{x | (x = d) \in D\}$, and by $fx(D)$ the associated valuation $\{x \mapsto d | (x = d) \in D\}$. Define $fixed(P) = fixed(D)$ and $fx(P) = fx(D)$ when $P \equiv (C, D)$.

A constraint $c \in C$ can be considered a set of valuations $solns(c)$ over the variables $vars(c)$. Valuation θ *satisfies* constraint c iff $vars(c) \subseteq vars(\theta)$ and $\theta_{vars(c)} \in c$. A *solution* of P is a valuation over $vars(P)$ that satisfies every constraint in C. We let $solns(P)$ be the set of all its solutions. Problem P is *satisfiable* if it has at least one solution and *unsatisfiable* otherwise.

Finally, we use $\exists_V.F$ to denote $\exists v_1.\exists v_2 \cdots \exists v_n.F$ where F is a formula and V is the set of variables $\{v_1, v_2, \ldots, v_n\}$. Similarly, we use $\bar{\exists}_V.F$ to denote the formula $\exists_{vars(F)-V}.F$. We let $\Leftrightarrow$ denote logical equivalence and $\Rightarrow$ logical entailment of formulae.

Given a constraint problem $P \equiv (C, D)$, constraint programming solves P by a search process that first uses a constraint solver to determine whether P can immediately be classified as satisfiable or unsatisfiable. We assume a propagation solver, denoted by solv, which when applied to P returns a new set D' of domain constraints such that $D' \Rightarrow D$ and $C \wedge D \Leftrightarrow C \wedge D'$. The solver

detects unsatisfiability if any $x \in \emptyset$ appears in D'. We assume that if the solver returns a domain D' where all variables are fixed ($fixed(D) = vars(D)$), then the solver has detected satisfiability of the problem and $fx(D)$ is a solution. If the solver cannot immediately determine whether P is satisfiable or unsatisfiable, the search splits P into n subproblems (obtained by adding one of $c_1, \ldots, c_n$ constraints to P, where $C \wedge D \models (c_1 \vee c_2 \vee \ldots \vee c_n)$) and iteratively searches for solutions to them.

The idea is for the search to drive towards subproblems that can be immediately detected by solv as being satisfiable or unsatisfiable. This solving process implicitly defines a *search tree* rooted by the original problem P where each node represents a new (though perhaps logically equivalent) subproblem P', which will be used as the node's label. For the purposes of this paper we restrict ourselves to the case where each c_i added by the search takes the form $x \in s$. This allows us to obtain the i-th subproblem from $P \equiv (C, D)$ and $c_i \equiv x \in s$ as simply $P_i \equiv (C, join(x, s, D))$, where $join(x, s, D)$ modifies the domain of x to be a subset of s: $join(x, s, D) = (D - \{x \in s_x\}) \cup \{x \in s \cap s_x\}$. While this is not a strong restriction, it does rule out some kinds of constraint programming search.

3 Problem Dominance and Equivalence

Consider two constraint problems $P \equiv (C, D)$ and $P' \equiv (C', D')$ and a set of variables U. Intuitively, we say that P U-dominates P' if variables not in U are fixed, and when P and P' are projected over U, the latter entails the former.

Definition 1. *(C, D) U-dominates (C', D') iff*

- $(vars(D) - U) \subseteq fixed(D)$ *and* $(vars(D') - U) \subseteq fixed(D')$, *and*
- $\bar{\exists}_U.(C' \wedge D') \Rightarrow \bar{\exists}_U.(C \wedge D)$.

Example 1. Consider $P_0 \equiv (C, D)$ where $C \equiv \{x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20\}$, $D \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$, and let $U \equiv \{x_3, x_4, x_5\}$. The subproblem $P \equiv (C, \{x_1 = 3, x_2 = 1\} \cup D_U)$ U-dominates $P' \equiv (C, \{x_1 = 1, x_2 = 3\} \cup D_U)$. □

If one problem P U-dominates another P' we can use the solutions of P to generate the solutions of P', as formalised by the following proposition.

Proposition 1. *If P U-dominates P' then $\theta \in solns(P)$ if $(\theta_U \cup fx(P')_{vars(P')-U}) \in solns(P')$.*

The situation is even simpler if the U-dominance relationship is symmetric.

Definition 2. *P and P' are U-equivalent iff P U-dominates P' and vice versa.*

Example 2. Consider problem (C, D) where $C \equiv \{alldiff([x_1, x_2, x_3, x_4, x_5])\}$ and $D \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$, and let $U \equiv \{x_3, x_4, x_5\}$. The subproblems $P \equiv (C, \{x_1 = 1, x_2 = 2\} \cup D_U)$ and $P' \equiv (C, \{x_1 = 2, x_2 = 1\} \cup D_U)$ are U-equivalent. □

```
cache_search(C, D)
  D' := solv(C, D)
  if (D' ⇔ false) return false
  if (∃U.∃P ∈ Cache where P U-dominates (C, D)) return false
  if fixed(D') ≡ vars(D)
    [SAT] return D
  foreach (x ∈ s) ∈ split(C, D)
    S := cache_search(C, join(x, s, D))
    if (S ≢ false) return S
  Cache := Cache ∪ {(C, D')}
  return false
```

Fig. 1. Computing the first solution under subproblem equivalence

Proposition 2. *If P and P' are U-equivalent then $\theta \in soln(P)$ iff $(\theta_U \cup fx(P')_{vars(P')-U}) \in soln(P')$.*

3.1 Searching with Caching

Detecting subproblem domination allows us to avoid exploring the dominated subproblem and reuse the solutions of the dominating subproblem (Proposition 1). This is particularly easy when we are only interested in the first solution, since we know the dominated subproblem must have no solutions. The algorithm for first solution satisfaction search using domination is shown in Figure 1. At each node, it propagates using solv. If it detects unsatisfiability it immediately fails. Otherwise, it checks whether the current subproblem is dominated by something already visited (and, thus, in $Cache$), and if so it fails, It then checks whether we have reached a solution and if so returns it. Otherwise it splits the current subproblem into a logically equivalent set of subproblems and examines each of them separately. When the entire subtree has been exhaustively searched, the subproblem is added to the cache.

The above algorithm can be straightforwardly extended to a branch and bound optimization search. This is because any subproblem cached has failed under a weaker set of constraints, and will thus also fail with a strictly stronger set of constraints. As a result, to extend the algorithm in Figure 1 to, for example, minimize the objective function $\sum_{i=1}^{n} a_i x_i$, we can simply replace the line labelled [SAT] by the following lines:[1]

```
globally store fx(D) as best solution
globally add Σ_{i=1}^{n} a_i x_i ≤ fx(D)(Σ_{i=1}^{n} a_i x_i) − 1
return false
```

Note that in this algorithm, the search always fails with the optimal solution being the last one stored.

[1] We assume there is an upper bound u on the objective function so that we can have a pseudo-constraint $\sum_{i=1}^{n} a_i x_i \leq u$ in the problem from the beginning, and replace it with the new one whenever a new solution is found.

4 Keys for Caching

The principal difficulty in implementing cache_search of Figure 1 is implementing the lookup and test for U-dominance. We need an efficient *key* to represent remaining subproblems that allows U-dominance to be detected efficiently (preferably in $O(1)$ time). Naively, one may think D would be a good key for subproblem $P \equiv (C, D)$. While using D as key is correct, it is also useless since D is different for each subproblem (i.e., node) in the search tree. We need to find a more general key; one that can represent equivalent subproblems with different domain constraints.

4.1 Projection Keys

We can automatically construct such a key by using constraint projection. Roughly speaking, subproblem U-equivalence arises whenever the value of some of the fixed variables in $C \wedge D$ and $C \wedge D'$ is different, but the global effect of the fixed variables on the unfixed variables of C is the same. Therefore, if we can construct a key that characterises exactly this effect, the key should be identical for all U-equivalent subproblems.

To do this, we need to characterize the projected subproblem of each $P \equiv (C, D)$ in terms of its projected variables and constraints. Let $F = fixed(D)$ and $U = vars(C) - F$. The projected subproblem can be characterized as:

$$\begin{aligned} & \bar{\exists}_U.(C \wedge D) \\ \Leftrightarrow\ & \bar{\exists}_U.(C \wedge D_F \wedge D_U) \\ \Leftrightarrow\ & \bar{\exists}_U.(C \wedge D_F) \wedge D_U \\ \Leftrightarrow\ & \bar{\exists}_U.(\wedge_{c \in C}(c \wedge D_F)) \wedge D_U \\ \Leftrightarrow\ & \wedge_{c \in C}(\bar{\exists}_U.(c \wedge D_F)) \wedge D_U \end{aligned}$$

The last step holds because all variables being projected out in every $c \wedge D_F$ were already fixed. Importantly, this allows each constraint $c \in C$ to be treated independently.

We can automatically convert this information into a key by *back projecting* the projected constraints of this problem to determine conditions on the fixed variables F. We define the back projection of constraint $c \in C$ for D_F as a constraint $BP(c, D_F)$ over variables $F \cap vars(c)$ such that $\bar{\exists}_U.(c \wedge BP(c, D_F)) \Leftrightarrow \bar{\exists}_U.(c \wedge D_F)$. Clearly, while $D_{F \cap vars(c)}$ is always a correct back projection, our aim is to define the most general possible back projection that ensures the equivalence. Note that if c has no variables in common with F, then $BP(c, D_F) \equiv true$. Note also that when c is implied by D_F, that is $\bar{\exists}_U.(c \wedge D_F) \Leftrightarrow true$, then c can be eliminated. We thus define $BP(c, D_F) \equiv red(c)$, where $red(c)$ is simply a name representing the disjunction of all constraints that force c to be redundant (we will see later how to remove these artificial constraints). The *problem key* for $P \equiv (C, D)$ is then defined as $key(C, D) \equiv \wedge_{c \in C} BP(c, D_F) \wedge D_U$.

Example 3. Consider the problem $C \equiv \{alldiff([x_1, x_2, x_3, x_4, x_5, x_6]), x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20\}$ and domain $D \equiv \{x_1 = 3, x_2 = 4, x_3 = 5, x_4 \in \{0, 1, 2\}, x_5 \in$

$\{0,1,2\}, x_6 \in \{1,2,6\}\}$. Then $F = \{x_1, x_2, x_3\}$ and $U = \{x_4, x_5, x_6\}$. The projected subproblem is characterized by $\mathit{alldiff}([x_4, x_5, x_6]) \wedge x_4 + 2x_5 \leq 4 \wedge D_U$. A correct back projection for *alldiff* onto $\{x_1, x_2, x_3\}$ is $\{x_1, x_2, x_3\} = \{3, 4, 5\}$. A correct back projection of the linear inequality is $x_1 + 2x_2 + x_3 = 16$. Thus, $key(C, D) \equiv \{x_1, x_2, x_3\} = \{3, 4, 5\} \wedge x_1 + 2x_2 + x_3 = 16 \wedge x_4 \in \{0, 1, 2\} \wedge x_5 \in \{1, 2\} \wedge x_6 \in \{1, 2, 6\}$. □

We now illustrate how to use the keys for checking dominance.

Theorem 1. *Let $P \equiv (C, D)$ and $P' \equiv (C, D')$ be subproblems arising during the search. Let $F = fixed(D)$ and $U = vars(C) - F$. If $fixed(D') = F$, $D'_U \Rightarrow D_U$ and $\forall c \in C.(\bar{\exists}_U.c \wedge BP(c, D'_F)) \Rightarrow (\bar{\exists}_U.c \wedge BP(c, D_F))$ then P U-dominates P'.*

Proof. The first condition of U-dominance holds since $vars(D') - U = F$. We show the second condition holds:

$$
\begin{aligned}
& \bar{\exists}_U.(C \wedge D') \\
\Leftrightarrow\ & \bar{\exists}_U.(C \wedge D'_F \wedge D'_U) \\
(\star) \Leftrightarrow\ & \wedge_{c \in C}(\bar{\exists}_U.c \wedge D'_F) \wedge D'_U \\
\Leftrightarrow\ & \wedge_{c \in C}(\bar{\exists}_U.c \wedge BP(c, D'_F)) \wedge D'_U \\
\Rightarrow\ & \wedge_{c \in C}(\bar{\exists}_U.c \wedge BP(c, D_F)) \wedge D_U \\
\Leftrightarrow\ & \wedge_{c \in C}(\bar{\exists}_U.c \wedge D_F) \wedge D'_U \\
(\star) \Leftrightarrow\ & \bar{\exists}_U.(C \wedge D_F \wedge D'_U) \\
\Rightarrow\ & \bar{\exists}_U.(C \wedge D)
\end{aligned}
$$

The second and sixth (marked) equivalences hold because, again, all variables being projected out in each $c \wedge D'_F$ and $c \wedge D_F$ were already fixed. □

Corollary 1. *Suppose $P \equiv (C, D)$ and $P' \equiv (C, D')$ are subproblems arising in the search tree for C. Let $F = fixed(D)$ and $U = vars(C) - F$. If $key(C, D) \equiv key(C, D')$ then $fixed(D') = F$ and P and P' are U-equivalent.*

Proof. Let $F' = fixed(D')$, $U' = vars(C) - F'$. Since $key(C, D) \equiv key(C, D')$ we have that $D_U \Leftrightarrow D'_{U'}$ and hence $F = F'$ and $U = U'$. Also clearly $\forall c \in C.BP(c, D_F) \equiv BP(c, D'_F)$ Hence, P U-dominates P' and vice versa. □

While determining a back projection is a form of *constraint abduction* which can be a very complex task, we only need to find simple kinds of abducibles for individual constraints and fixed variables. Hence, we can define for each constraint a method to determine a back projection. Figure 2 shows back projections for some example constraints and variable fixings.

Note that a domain consistent binary constraint c always has either no unfixed variables (and, hence, its back projection is *true*), or all its information is captured by domain constraints (and, hence, it is redundant and its back projection is $red(c)$).

constraint c	D_F	$\exists_U.c$	$BP(c, D_F)$
$alldiff([x_1, \ldots, x_n])$	$\wedge_{i=1}^{m} x_i = d_i$	$alldiff([d_1, \ldots, d_m, x_{m+1}, \ldots x_n])$	$\{x_1, \ldots, x_m\} = \{d_1, \ldots, d_m\}$
$\sum_{i=1}^{n} a_i x_i = a_0$	$\wedge_{i=1}^{m} x_i = d_i$	$\sum_{i=m+1}^{n} a_i x_i = a_0 - \sum_{i=1}^{m} a_i d_i$	$\sum_{i=1}^{m} a_i x_i = \sum_{i=1}^{m} a_i d_i$
$\sum_{i=1}^{n} a_i x_i \leq a_0$	$\wedge_{i=1}^{m} x_i = d_i$	$\sum_{i=m+1}^{n} a_i x_i \leq a_0 - \sum_{i=1}^{m} a_i d_i$	$\sum_{i=1}^{m} a_i x_i = \sum_{i=1}^{m} a_i d_i$
$x_0 = \min_{i=1}^{n} x_i$	$\wedge_{i=1}^{m} x_i = d_i$ $x_0 = d_0$	$x_o = \min(\min_{i=1}^{m} d_i, \min_{i=m+1}^{n} x_i)$ $\wedge_{i=1}^{n} x_i \geq d_0 \wedge \vee_{i=1}^{n} x_i = d_0$	$\min_{i=1}^{m} x_i = \min_{i=1}^{m} d_i$ $x_0 = d_0$
$\vee_{i=1}^{n} x_i$	$x_1 = true$ $\wedge_{i=1}^{m} x_i = false$	$true$ $\vee_{i=m+1}^{n} x_i$	$red(\vee_{i=1}^{n} x_i)$ $\wedge_{i=1}^{m} x_i = false$

Fig. 2. Example constraints with their fixed variables, projections and resulting back projection

4.2 Using Projection Keys

By Corollary 1, if we store every explored subproblem P in the cache using $key(P)$, and we encounter a subproblem P' such that $key(P')$ appears in the cache, then P' is equivalent to a previous explored subproblem and does not need to be explored.

Example 4. Consider the problem $P \equiv (C, D)$ of Example 3 and the new subproblem $P' \equiv (C, D')$ where $D' \equiv \{x_1 = 5, x_2 = 4, x_3 = 3, x_4 \in \{0, 1, 2\}, x_5 \in \{0, 1\}, x_6 \in \{1, 2, 6\}\}$. The characterisation of the projected subproblem for P' is identical to that obtained in Example 3 and, hence, $key(P) \equiv key(P')$ indicating P and P' are U-equivalent. □

If we are using projection keys for detecting subproblem equivalence, we are free to represent the keys in any manner that illustrates identity. This gives use the freedom to generate space efficient representations, and choose representations for $BP(c, D_F)$ on a per constraint basis.

Example 5. Consider the problem $P \equiv (C, D)$ of Example 3. We can store its projection key $\{x_1, x_2, x_3\} = \{3, 4, 5\} \wedge x_1 + 2x_2 + x_3 = 16 \wedge D_U$ as follows: We store the fixed variables $\{x_1, x_2, x_3\}$ for the subproblem since these must be identical for the equivalence check in any case. We store $\{3, 4, 5\}$ for the *alldiff* constraint, and the fixed value 16 for the linear constraint, which give us enough information given the fixed variables to define the key. The remaining part of the key are domains. Thus, the projection key can be stored as $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 16, \{0, 1, 2\}, \{0, 1, 2\}, \{1, 2, 6\})$ □

Theorem 1 shows how we can make use of projection keys to determine subproblem dominance. If we store $key(P)$ in the cache we can determine if new subproblem P' is dominated by a previous subproblem by finding a key where the fixed variables are the same, each projection of a primitive constraint for P' is at least as strong as the projection defined by $key(P)$, and the domains of the unfixed variables in P' are at least as strong as the unfixed variables in $key(P)$.

Example 6. Consider $P \equiv (C, D)$ of Example 3 and the new subproblem $P' \equiv (C, D')$ where $D' \equiv \{x_1 = 4, x_2 = 5, x_3 = 3, x_4 \in \{0, 1, 2\}, x_5 \in \{0, 1\}, x_6 \in \{1, 2, 6\}\}$. We have that $fixed(D') = fixed(D) = \{x_1, x_2, x_3\}$ and the back projections of the *alldiff* are identical. Also, the projection of the linear inequality is

$x_4+2x_5 \leq 3$. This is stronger than the projection in $key(P)$ which is computable as $\exists x_1.\exists x_2.\exists x_3.x_1+2x_2+x_3 = 16 \wedge x_1+2x_2+x_3+x_4+2x_5 \leq 20 \Leftrightarrow x_4+2x_5 \leq 4$. Similarly, $D'_U \Rightarrow D_U$. Hence, P $\{x_4, x_5, x_6\}$-dominates P'. □

To use projection keys for dominance detection we need to check $D'_U \Rightarrow D_U$ and $(\bar{\exists}_U.c \wedge BP') \Rightarrow (\bar{\exists}_U.c \wedge BP)$. Note that if $BP \equiv red(c)$, then the entailment automatically holds and we do not need to store these artificial projection keys. Note also that we can make the choice of how to check for entailment differently for each constraint. We will often resort to identity checks as a weak form of entailment checking, since we can then use hashing to implement entailment.

Example 7. Consider $P \equiv (C, D)$ of Example 3. Entailment for *alldiff* is simply identity on the set of values, while for the linear constraint we just compare fixed values, since $(\exists x_1.\exists x_2.\exists x_3.\ x_1+2x_2+x_3 = k \wedge x_1+2x_2+x_3+x_4+2x_5 \leq 20) \Leftrightarrow x_4+2x_5 \leq 20-k \Rightarrow (\exists x_1.\exists x_2.\exists x_3.\ x_1+2x_2+x_3 = k' \wedge x_1+2x_2+x_3+x_4+2x_5 \leq 20) \Leftrightarrow x_4+2x_5 \leq 20-k'$ whenever $k \geq k'$. For the problem P' of Example 6 we determine the key $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 17, \{0, 1, 2\}, \{0, 1\}, \{1, 2, 6\})$. We can hash on the first two arguments of the tuple to retrieve the key for P, and then compare 17 versus 16 and check that each of the three last arguments is a superset of that appearing in $key(P')$. Hence, we determine the dominance holds. □

Note that, for efficiency, our implementation checks $D'_U \Rightarrow D_U$ by using identity $(D'_U \equiv D_U)$ so the domains can be part of the hash value. This means that the problem P' of Example 6 will not be detected as dominated in our implementation, since the domain of x_5 is different.

4.3 Caching Optimal Subproblem Values

The presentation so far has concentrated on satisfaction problems; let us examine what happens with optimization problems. Typically, when solving optimization problems with caching one wants to store optimal partial objective values with already explored subproblems. We shall see how our approach effectively manages this automatically using dominance detection with a minor change.

Suppose k is the current best solution found. Then, the problem constraints must include $\sum_{i=1}^{n} a_i x_i \leq k-1$ where $\sum_{i=1}^{n} a_i x_i$ is the objective function. Suppose we reach a subproblem $P \equiv (C, D)$ where $D_{fixed(D)} \equiv \{x_1 = d_1, \ldots, x_m = d_m\}$ are the fixed variables. The remaining part of the objective function constraint is $\sum_{i=m+1}^{n} a_i x_i \leq k-1-p$ where $p = \sum_{i=1}^{m} a_i d_i$, and the back projection is $\sum_{i=1}^{m} a_i x_i = p$. The projection key contains the representation p for this back projection. If this subproblem fails we have proven that, with D, there is no solution with a value $< k$, nor with $\sum_{i=m+1}^{n} a_i x_i \leq k-1-p$.

If we later reach a subproblem $P' \equiv (C, D')$ where $D' \Rightarrow x_1 = d'_1 \wedge \cdots \wedge x_m = d'_m$ are the fixed variables, then dominance requires $p' = \sum_{i=1}^{m} a_i d'_i$ to satisfy $p' \geq p$. If this does not hold it may be that a solution for the projected problem with $\sum_{i=m+1}^{n} a_i x_i \geq k-p$ can lead to a global solution $< k$. Hence, we do have to revisit this subproblem.

Suppose that by the time we reach P', a better solution $k' < k$ has been discovered. Effectively the constraint $\sum_{i=1}^{n} a_i x_i \leq k - 1$ has been replaced by $\sum_{i=1}^{n} a_i x_i \leq k' - 1$. Now we are only interested in finding a solution where $\sum_{i=m+1}^{n} a_i x_i \leq k' - 1 - p'$. To see if this is dominated by a previous subproblem, the stored value p is not enough. We also need the optimal value k when the key was stored. There is a simple fix: rather than storing p in the key for P we store $q = k - 1 - p$. We can detect dominance if $q \leq k' - 1 - p'$ and this value q is usable for all future dominance tests. Note that q *implicitly* represents the partial objective bound on the subproblem P.

5 Related Work

Problem specific approaches to dominance detection/subproblem equivalance are widespread in combinatorial optimization (see e.g. [6,19]) There is also a significant body of work on caching that rely on problem decomposition by fixing variables (e.g [10,12]). This work effectively looks for equivalent projected problems, but since they do not take into account the semantics of the constraints, they effectively use $D_{F \cap vars(c)}$ for every constraint c as the projection key, which finds strictly fewer equivalent subproblems than back-projection. The success of these approaches in finding equivalent subproblems relies on decomposing the projected subproblem into disjoint parts. We could extend our approach to also split the projected problem into connected components but this typically does not occur in the problems of interest to us. Interestingly, [10] uses symmetry detection to make subproblem equivalence detection stronger, but the method used does not appear to scale.

5.1 Dynamic Programming

Dynamic programming (DP) [2] is a powerful approach for solving optimization problems whose optimal solutions are derivable from the optimal solutions of its subproblems. It relies on formulating an optimization as recursive equations relating the answers to optimization problems of the same form. When applicable, it is often near unbeatable by other optimization approaches.

Constraint programming (CP) with caching is similar to DP, but provides several additional capabilities. For example, arbitrary side constraints not easily expressible as recursions in DP can easily be expressed in CP, and dominance can be expressed and exploited much more naturally in CP.

Consider the 0-1 Knapsack problem, a well known NP-hard problem that is easy to formulate using recursive equations suitable for DP. We show how our automatic caching provides a different but similar solution, and how caching can change the asymptotic complexity of the CP solution. The problem is to maximise $\sum_{i=1}^{n} p_i x_i$ subject to the constraints $\sum_{i=1}^{n} w_i x_i \leq W \wedge \forall_{i=1}^{n} x_i \in \{0, 1\}$, where w_i is the nonnegative weight of object i and p_i is the nonnegative profit. A normal CP solver will solve this problem in $O(2^n)$ steps.

The DP formulation defines $knp(j, w)$ as the maximum profit achievable using the first j items with a knapsack of size w. The recursive equation is

$$knp(j,w) = \begin{cases} 0 & j = 0 \vee w \leq 0 \\ \max(knp(j-1,w), knp(j-1,w-w_j)+p_j) & \text{otherwise} \end{cases}$$

The DP solution is $O(nW)$ since values for $knp(j, w)$ are cached and only computed once. Consider a CP solver using a fixed search order $x_1, \ldots, x_n$. A subproblem fixing $x_1, \ldots, x_m$ to $d_1, \ldots, d_m$ respectively generates key value $\sum_{i=1}^{m} w_i d_i$ for the constraint $\sum_{i=1}^{n} w_i x_i \leq W$ and key value $k+1-\sum_{i=1}^{m} p_i d_i$ for the optimization constraint $\sum_{i=1}^{n} p_i x_i \geq k+1$ where k is the best solution found so far. The remaining variable domains are all unchanged so they do not need to be explicitly stored (indeed domains of Boolean or 0-1 variables never need to be stored as they are either fixed or unchanged). The projection key is simply the set of fixed variables $\{x_1, \ldots, x_m\}$ and the two constants. The complexity is hence $O(nWu)$ where u is the initial upper bound on profit.

The solutions are in fact quite different: the DP approach stores the optimal profit for each set of unfixed variables and remaining weight limit, while the CP approach stores the fixed variables and uses weight plus the remaining profit required. The CP approach in fact implements a form of DP with bounding [16]. In particular, the CP approach can detect subproblem dominance, a problem with used weight w' and remaining profit required p' is dominated by a problem with used weight $w \leq w'$ and remaining profit $p \leq p'$. The DP solution must examine both subproblems since the remaining weights are different.

In practice the number of remaining profits arising for the same set of fixed variables and used weight is $O(1)$ and hence the practical number of subproblems visited by the CP approach is $O(nW)$.

Note that while adding a side constraint like $x_3 \geq x_8$ destroys the DP approach (or at least forces it to be carefully reformulated), the CP approach with automatic caching works seamlessly.

5.2 Symmetry Breaking

Symmetry breaking aims at speeding up execution by not exploring search nodes known to be symmetric to nodes already explored. Once the search is finished, all solutions can be obtained by applying each symmetry to each solution. In particular, Symmetry Breaking by Dominance Detection (SBDD) [4] works by performing a "dominance check" at each search node and, if the node is found to be dominated, not exploring the node.

SBDD is related but different to automatic caching. In SBDD $P \equiv (C, D)$ ϕ-dominates $P' \equiv (C, D')$ under symmetry ϕ iff $\phi(D') \Rightarrow D$ since, if this happens, the node associated to symmetric problem $(C, \phi(D'))$ must be a descendant of the node associated to P and, thus, already explored. Note that, in detecting dominance, SBDD places conditions on the domains of *all* variables in D, while automatic caching only does so on the constraints of the problem *once projected on the unfixed variables*. Thus, P' can be ϕ-dominated by P (in the SBDD sense) but not be U-dominated, and vice versa.

Our approach is also related to conditional symmetry breaking [7], which identifies conditions that, when satisfied in a subproblem, ensure new symmetries occur within that subproblem (and not in the original problem). As before, the two approaches capture overlapping but distinct sets of redundancies.

5.3 Nogood Learning

Nogood learning approaches in constraint programming attempt to learn from failures and record these as new constraints in the program. The most successful of these methods use clauses on atomic constraints $v = d$ and $v \leq d$ to record the reasons for failures and use SAT techniques to efficiently manage the nogoods. Automatic caching is also a form of nogood learning, since it effectively records keys that lead to failure.

Any nogood learning technique representing nogoods as clauses has the advantage over caching that it can use the nogoods derived to propagate rather than to simply fail. Restart learning [11] simply records failed subtrees using the set of decisions made to arrive there. This does not allow subproblem equivalence to be detected assuming a fixed search strategy. The usefulness arises because it is coupled with restarting and dynamic search, so it helps avoid repeated search. In that sense it has a very different aim to automatic caching. Nogood learning techniques such as lazy clause generation [14] learn clauses that are derived only from the constraints that are actually involved in conflicts, which is much more accurate than using all non-redundant constraints as in projection keys.

On the other hand nogood learning can come at a substantial price: reason generation and conflict analysis can be costly. Every clause learnt in a nogood approach adds extra constraints and, hence, slows down the propagation of the solver. In contrast, projection keys are $O(1)$ to lookup regardless of their number (at least for the parts that are in the hash).

Because nogood learning use clauses on atomic constraints to define nogoods they may be less expressive than projection keys. Consider the subproblem $x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20 \wedge C$, with $D \equiv \{x_1 = 1 \wedge x_2 = 2 \wedge x_3 = 3\}$. If this subproblem fails, the projection key stores that $x_4 + 2x_5 \leq 12 \wedge$other keys leads to failure. A nogood system will express this as $x_1 = 1 \wedge x_2 = 2 \wedge x_3 = 3 \wedge$other keys leads to failure, since there are no literals to representing partial sums. This weakness is illustrated by the experimental results for 0-1 Knapsack.

6 Experiments

We compare our solver Chuffed, with and without caching, against Gecode 3.2.2 [17] – widely recognized as one of the fastest constraint programming systems (to illustrate we are not optimizing a slow system) – against the G12 FD solver [15] and against the G12 lazy clause generation solver [5] (to compare against nogood learning). We use the `MurmurHash 2.0` hash function. We use models written in the modelling language MiniZinc [13]. This facilitates a fair comparison between the solvers, as all solvers use the same model and search

strategy. Note that caching does not interfere with the search strategies used here, as all it can do is fail subtrees earlier. Thus, CHUFFED with caching (denoted as CHUFFEDC) always finds the same solution as the non-caching version and the other solvers, and any speedup observed comes from a reduced search.

Considerable engineering effort has gone into making caching as efficient as possible: defining as general as possible back projections for each of the many primitive constraints defined in the solver, exploiting small representations for back projections and domains, and eliminating information that never needs storing, e.g. binary domain consistent constraints and Boolean domains.

The experiments were conducted on Xeon Pro 2.4GHz processors with a 900 second timeout. Table 1 presents the number of variables and constraints as reported by CHUFFED, the times for each solver in seconds, and the speedup and node reduction obtained from using automatic caching in CHUFFED. We discuss the results for each problem below. All the MiniZinc models and instances are available at `www.cs.mu.oz.au/~pjs/autocache/`

Knapsack. 0-1 knapsack is ideal for caching. The non-caching solvers all timeout as n increases, as their time complexity is $O(2^n)$. This is a worst case for lazy clause generation since the nogoods generated are not reusable. CHUFFEDC, on the other hand, is easily able to solve much larger instances (see Table 1). The node to nW ratio (not shown) stays fairly constant as n increases (varying between 0.86 and 1.06), showing that it indeed has search (node) complexity $O(nW)$. The time to nW ratio grows as $O(n)$ though, since we are using a general CP solver where the linear constraints take $O(n)$ to propagate at each node, while DP requires constant work per node. Hence, we are not as efficient as pure DP.

MOSP. The minimal open stacks problem (MOSP) aims at finding a schedule for manufacturing all products in a given set that minimizes the maximum number of active customers, i.e., the number of customers still waiting for at least one of their products to be manufactured. This problem was the subject of the 2005 constraint modelling challenge [18]. Of the 13 entrants only 3 made use of the subproblem equivalence illustrating that, in general, it may not be easy to detect. Our MOSP model uses customer search and some complex conditional dominance breaking constraints that make the (non-caching) search much faster. We use random instances from [3]. Automatic caching gives up to two orders of magnitude speedup. The speedup grows exponentially with problem size. Lazy clause is also capable of exploiting this subproblem equivalence, but the overhead is so large that it can actually slow the solver down.

Blackhole. In the Blackhole patience game, the 52 cards are laid out in 17 piles of 3, with the ace of spades starting in a "blackhole". Each turn, a card at the top of one of the piles can be played into the blackhole if it is +/-1 from the card that was played previously. The aim is to play all 52 cards. This was one of two examples used to illustrate CP with caching in [19]. The remaining subproblem only depends on the set of unplayed cards, and the value of the last card played.

Thus, there is subproblem equivalence. We use a model from [7] which includes conditional symmetry breaking constraints. We generated random instances and used only the hard ones for this experiment. The G12 solvers do not use a domain consistent *table* constraint for this problem and are several orders of magnitudes slower. Automatic caching gives a modest speedup of around 2-3. The speedup is relatively low on this problem because the conditional symmetry breaking constraints have already removed many equivalent subproblems, and the caching is only exploiting the ones which are left. Note that the manual caching reported in [19] achieves speedups in the same range (on hard instances).

BACP. In the Balanced Academic Curriculum Problem (BACP), we form a curriculum by assigning a set of courses to a set of periods, with certain restrictions on how many courses and how much "course load" can be assigned to each period. We also have prerequisite constraints between courses. The BACP can be viewed as a bin packing problem with a lot of additional side constraints. The remaining subproblem only depends on the set of unassigned courses, and not on how the earlier courses were assigned. We use the model of [9], but with some additional redundant constraints that make it very powerful. The 3 instances *curriculum_*8/10/12 given in CSPLIB can be solved to optimality in just a few milliseconds. We generate random instances with 50 courses, 10 periods, and course credit ranging between 1 and 10. Almost all are solvable in milliseconds so we pick out only the non-trivial ones for the experiment. We also include the 3 standard instances from CSPLIB. Both automatic caching and lazy clause generation are capable of exploiting the subproblem equivalence, giving orders of magnitude speedup. In this case, lazy clause generation is more efficient.

Radiation Therapy. In the Radiation Therapy problem [1], the aim is to decompose an integral intensity matrix describing the radiation dose to be delivered to each area, into a set of patterns to be delivered by a radiation source, while minimising the amount of time the source has to be switched on, as well as the number of patterns used (setup time of machine). The subproblem equivalence arises because there are equivalent methods to obtain the same cell coverages, e.g. radiating one cell with two intensity 1 patterns is the same as radiating it with one intensity 2 pattern, etc. We use random instances generated as in [1]. Both automatic caching and lazy clause generation produce orders of magnitude speedup, though lazy clause generation times are often slightly better.

Memory Consumption. The memory consumption of our caching scheme is linear in the number of nodes searched. The size of each key is dependent on the structure of the problem and can range from a few hundred bytes to tens of thousands of bytes. On a modern computer, this means we can usually search several hundreds of thousands of nodes before running out of memory. There are simple schemes to reduce the memory usage, which we plan to investigate in the future. For example, much like in SAT learning, we can keep an "activity" score for each entry to keep track of how often they are used. Inactive entries can then periodically be pruned to free up memory.

Table 1. Experimental Results

Instance	vars	cons.	ChuffedC	Chuffed	Gecode	G12_fd	G12_lazyfd	Speedup	Node red.
knapsack-20	21	2	0.01	0.01	0.01	0.01	0.10	1.00	2.9
knapsack-30	31	2	0.02	0.83	0.76	1.168	534.5	41.5	67
knapsack-40	41	2	0.03	38.21	34.54	58.25	>900	1274	1986
knapsack-50	51	2	0.07	>900	>900	>900	>900	>12860	>20419
knapsack-60	61	2	0.10	>900	>900	>900	>900	>9000	>14366
knapsack-100	101	2	0.40	>900	>900	>900	>900	>2250	> 2940
knapsack-200	201	2	2.36	>900	>900	>900	>900	>381	> 430
knapsack-300	301	2	6.59	>900	>900	>900	>900	>137	>140
knapsack-400	401	2	13.96	>900	>900	>900	>900	>65	>65
knapsack-500	501	2	25.65	>900	>900	>900	>900	>35	> 34
mosp-30-30-4-1	1021	1861	1.21	4.80	24.1	50.29	29.70	4.0	4.91
mosp-30-30-2-1	1021	1861	6.24	>900	>900	>900	201.8	>144	>187
mosp-40-40-10-1	1761	3281	0.68	0.66	5.85	15.07	29.80	1.0	1.1
mosp-40-40-8-1	1761	3281	1.03	1.15	9.92	27.00	56.96	1.1	1.3
mosp-40-40-6-1	1761	3281	3.79	11.30	75.36	183.9	165.2	3.0	3.5
mosp-40-40-4-1	1761	3281	19.07	531.68	>900	>900	840.4	28	37
mosp-40-40-2-1	1761	3281	60.18	>900	>900	>900	>900	>15	> 18
mosp-50-50-10-1	2701	5101	2.83	3.17	40.70	92.74	134.1	1.1	1.2
mosp-50-50-8-1	2701	5101	6.00	9.12	113.0	292.0	295.9	1.5	1.8
mosp-50-50-6-1	2701	5101	39.65	404.16	>900	>900	>900	10.2	13.1
blackhole-1	104	407	18.35	39.77	103.6	>900	>900	2.17	2.90
blackhole-2	104	411	14.60	21.52	60.06	>900	>900	1.47	1.94
blackhole-3	104	434	18.31	26.14	31.43	>900	>900	1.43	1.81
blackhole-4	104	393	15.77	30.84	69.13	>900	>900	1.96	2.55
blackhole-5	104	429	24.88	58.77	159.5	>900	>900	2.36	3.45
blackhole-6	104	448	11.31	33.27	85.65	>900	>900	2.94	5.11
blackhole-7	104	407	28.02	47.31	127.6	>900	>900	1.69	2.49
blackhole-8	104	380	24.09	43.60	89.02	>900	>900	1.81	2.45
blackhole-9	104	404	38.74	93.92	215.1	>900	>900	2.42	3.52
blackhole-10	104	364	67.85	159.4	418.0	>900	>900	2.35	3.16
curriculum_8	838	1942	0.01	0.01	0.01	0.02	0.08	1.00	1.00
curriculum_10	942	2214	0.01	0.01	0.01	0.03	0.09	1.00	1.00
curriculum_12	1733	4121	0.01	0.01	0.01	0.10	0.23	1.00	1.00
bacp-medium-1	1121	2654	11.47	34.90	29.31	62.4	6.90	3.04	3.03
bacp-medium-2	1122	2650	9.81	>900	>900	>900	0.22	>92	>115
bacp-medium-3	1121	2648	2.42	380.7	461.62	838.6	0.23	157	190
bacp-medium-4	1119	2644	0.61	4.59	5.74	9.92	1.10	7.52	10.1
bacp-medium-5	1119	2641	2.40	56.46	54.03	126.9	0.76	23.5	26.5
bacp-hard-1	1121	2655	54.66	>900	>900	>900	0.16	>16	>16
bacp-hard-2	1118	2651	181.9	>900	>900	>900	0.22	>5	>7
radiation-6-9-1	877	942	12.67	>900	>900	>900	2.89	>71	>146
radiation-6-9-2	877	942	27.48	>900	>900	>900	5.48	>32	>86
radiation-7-8-1	1076	1168	0.84	>900	>900	>900	1.40	>1071	>5478
radiation-7-8-2	1076	1168	0.65	89.18	191.4	173.6	0.93	137	633
radiation-7-9-1	1210	1301	2.39	143.0	315.6	241.9	2.70	59	266
radiation-7-9-2	1210	1301	7.26	57.44	144.4	101.9	8.83	8	34
radiation-8-9-1	1597	1718	27.09	>900	>900	>900	6.21	>33	>114
radiation-8-9-2	1597	1718	12.21	>900	>900	>900	6.53	>74	>267
radiation-8-10-1	1774	1894	22.40	12.17	15.45	12.90	33.2	0.54	1.10
radiation-8-10-2	1774	1894	59.66	>900	>900	>900	12.05	>15	>78

7 Conclusion

We have described how to automatically exploit subproblem equivalence in a general constraint programming system by automatic caching. Our automatic caching can produce orders of magnitude speedup over our base solver CHUFFED, which (without caching) is competitive with current state of the art constraint programming systems like Gecode. With caching, it can be much faster on problems that have subproblem equivalences.

The automatic caching technique is quite robust. It can find and exploit subproblem equivalence even in models that are not "pure", e.g. MOSP with dominance and conditional symmetry breaking constraints, Blackhole with conditional symmetry breaking constraints, and BACP which can be seen as bin packing with lots of side constraints and some redundant constraints. The speedups from caching tends to grow exponentially with problem size/difficulty, as subproblem equivalences also grow exponentially.

Our automatic caching appears to be competitive with lazy clause generation in exploiting subproblem equivalence, and is superior on some problems, in particular those with large linear constraints.

The overhead for caching is quite variable (it can be read from the tables as the ratio of node reduction to speedup). For large problems with little variable fixing it can be substantial (up to 5 times for radiation), but for problems that fix variables quickly it can be very low. Automatic caching of course relies on subproblem equivalence occurring to be of benefit. Note that for dynamic searches this is much less likely to occur. Since it is trivial to invoke, it seems always worthwhile to try automatic caching for a particular model, and determine empirically if it is beneficial.

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. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 1–15. Springer, Heidelberg (2007)
2. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
3. Chu, G., Stuckey, P.J.: Minimizing the maximum number of open stacks by customer search. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 242–257. Springer, Heidelberg (2009)
4. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 93–107. Springer, Heidelberg (2001)
5. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 352–366. Springer, Heidelberg (2009)
6. Fukunaga, A., Korf, R.: Bin completion algorithms for multicontainer packing, knapsack, and covering problems. J. Artif. Intell. Res. (JAIR) 28, 393–429 (2007)

7. Gent, I., Kelsey, T., Linton, S., McDonald, I., Miguel, I., Smith, B.: Conditional symmetry breaking. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 256–270. Springer, Heidelberg (2005)
8. Gent, I., Petrie, K., Puget, J.-F.: Symmetry in Constraint Programming. In: Handbook of Constraint Programming, pp. 329–376. Elsevier, Amsterdam (2006)
9. Hnich, B., Kiziltan, Z., Walsh, T.: Modelling a balanced academic curriculum problem. In: Proceedings of CPAIOR 2002, pp. 121–131 (2002)
10. Kitching, M., Bacchus, F.: Symmetric component caching. In: Proceedings of IJCAI 2007, pp. 118–124 (2007)
11. Lynce, I., Baptista, L., Marques-Silva, J.: Complete search restart strategies for satisfiability. In: IJCAI Workshop on Stochastic Search Algorithms, pp. 1–5 (2001)
12. Marinescu, R., Dechter, R.: And/or branch-and-bound for graphical models. In: Proceedings of IJCAI 2005, pp. 224–229 (2005)
13. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
14. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
15. Stuckey, P.J., Garcia de la Banda, M., Maher, M.J., Marriott, K., Slaney, J.K., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 13–16. Springer, Heidelberg (2005)
16. Puchinger, J., Stuckey, P.J.: Automating branch-and-bound for dynamic programs. In: Proceedings of PEPM 2008, pp. 81–89 (2008)
17. Schulte, C., Lagerkvist, M., Tack, G.: Gecode, `http://www.gecode.org/`
18. Smith, B., Gent, I.: Constraint modelling challenge report 2005 (2005), `http://www.cs.st-andrews.ac.uk/~ipg/challenge/ModelChallenge05.pdf`
19. Smith, B.M.: Caching search states in permutation problems. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 637–651. Springer, Heidelberg (2005)