Exploiting Subproblem Dominance 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 *subproblem dominance*, which arises when different partial assignments lead to subproblems that dominate (or are dominated by) other subproblems. Subproblem dominance is exploited by caching subproblems that have already been explored by the search, using keys that characterise the subproblems, and failing the search when the current subproblem is dominated by a subproblem already in the cache. In this paper we show how we can automatically and efficiently define keys for arbitrary constraint problems using constraint projection. We show how, for search problems where subproblem dominance arises, a constraint programming solver with this capability can solve these problems orders of magnitude faster than solvers without caching. The system is fully automatic, i.e., subproblem dominance is detected and exploited without any effort from the problem modeller.

1 Introduction

When solving a combinatorial search problem, it is common for the search to do redundant work due to the existence of different search paths leading to subproblems whose information can be somehow re-used by other subproblems. For example, the solutions and failures to subproblem P can be re-used by subproblem P' if P and P' satisfy certain properties (e.g., if they are equivalent, if they are symmetric, or if P' is dominated by P). There are a number of different methods that avoid (or reduce) redundant search, including caching solutions (e.g. [20]), symmetry breaking (e.g. [9]), and nogood learning (e.g. [15]).

This paper focuses on caching, which works by storing information in a cache regarding every new subproblem explored during the search. Right before a subproblem is explored, the search performs a lookup on the cache to check whether it contains an already explored subproblem whose information (such as failure, solutions or a bound on the objective function) can be used for the current subproblem. If so, the search does not explore the current subproblem and, instead, uses the stored information. Otherwise, the search continues exploring the subproblem and, once this is done, it stores the appropriate subproblem information in the cache.

For caching to be useful, the lookup operation must be efficient. A popular way to implement this operation is to map each subproblem to a *key* which is then used to store and access the information for that subproblem. The mapping is done in such a way that any test for re-usability can be performed on the subproblem keys rather than on the subproblems themselves. In the case of subproblem equivalence (which is the most restrictive form of re-usability and, thus, simplest to detect) the subproblems are often mapped to the same key and, thus, the test is a simple equality. Other kinds of re-usability might require more complex tests, such as logical entailment.

This paper explores how to use caching *automatically* to avoid redundancy in constraint programming (CP) search. While caching has been previously used in CP search, it has either relied on the careful manual construction of the key for each model and search strategy (e.g. [20]), or on being able to decompose the problem into independent components by fixing some of its variables (e.g. [11, 13]). 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 re-usability that we will refer to as subproblem dominance. The key is obtained by simply projecting each primitive constraint onto the set of subproblem variables not yet fixed to a particular value. We experimentally demonstrate the effectiveness of our approach, which has been implemented in the competitive CP solver CHUFFED. We also provide interesting insights into the relationships between subproblem dominance and dynamic programming, symmetry breaking and nogood learning.

The rest of the paper is organized as follows. The next section provides the necessary background for constraint programming and the notation that will be used throughout the paper. Section 3 defines subproblem dominance and subproblem equivalence, and shows how we can make use of these notions to improve search by caching. Section 4 gives some examples of manual keys to support caching of specific models with specific search strategies. Section 5 shows how we can define keys for arbitrary problems on a per constraint basis and taking into account the propagation algorithm of the constraint. Section 6 shows how to construct keys and efficient representations of keys for common primitive and global constraints. Section 7 illustrates the keys generated by automatic caching on a number of example problems. Section 8 discusses earlier work on caching in constraint programming and the relationship with related approaches such as dynamic programming and symmetry elimination. Section 9 provides a experimental evaluation showing that automatic caching can be very effective when subproblem equivalence exists. Finally, Section 10 provides our conclusions.

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 $D : x \not\equiv y$. A *false domain* is one where $(x \in \emptyset) \in D$ for some variable x. The *restriction* of D to variables V is the set of domain constraints $D_V = \{(x \in S_x) \in D | x \in V\}$. Each set D and C is logically interpreted as the conjunction of its elements. Note that a false domain D is logically equivalent to *false*. We use $D(x)$ to denote the set of possible values for x in domain D, that is, the set S_x where $(x \in S_x) \in D$.

Given a constraint problem $P \equiv (C, D)$, a *literal* of P is of the form $x \mapsto d$, where $d \in D(x)$. 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. In other words, 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 define the set of *fixed* variables in D as $fixed(D) = \{x | (x = d) \in D\}$ and the associated fixed valuation as $fx(D) = \{x \mapsto d | (x = d) \in D\}$. We also define $fixed(P)$ as $fixed(D)$ and $fx(P)$ as $fx(D)$.

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 $\theta_{vars(c)} \in solns(c)$. 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 and say that problem P is *satisfiable* if it has at least one solution and *unsatisfiable* otherwise.

We use $\Rightarrow$ to denote logical entailment and $\Leftrightarrow$ to denote logical equivalence. Given a logical formula F and a set of variables $V \equiv \{v_1, \ldots, v_n\}$ where $V \subseteq vars(F)$, we use $\exists_V.F$ to denote the existential quantification of every variable in V in formula F, i.e., $\exists v_1. \ldots .\exists v_n.F$.

Existential quantification effectively projects constraints onto a subset of their variables. For example, the projection of *alldiff*$([x_1, x_2, x_3, x_4])$ onto the subset of variables $\{x_3, x_4\}$ is: $\exists x_1.\exists x_2.$*alldiff*$([x_1, x_2, x_3, x_4])$ which is equivalent to *alldiff*$([x_3, x_4])$. The projection of $(x_1 = 1 \wedge x_1 + x_2 + x_3 \geq 10)$ onto $\{x_2, x_3\}$ is: $\exists x_1.(x_1 = 1 \wedge x_1 + x_2 + x_3 \geq 10)$ which is equivalent to $x_2 + x_3 \geq 9$. Note that it is perfectly possible to existentially quantify a variable which is fixed by the constraint (as in the second example above).

In constraint programming systems each constraint c is implemented by a *propagator* p_c that maps domains to domains. For correctness, we require that $c \wedge D \Leftrightarrow c \wedge p_c(D)$ for any propagator p_c implementing c. We assume each propagator is *checking*, i.e., it is strong enough to decide the satisfiability of constraint c when all the variables in c are fixed by the domain. Formally, we say that a propagator p_c is checking if $fixed(D) \supseteq vars(c)$ implies $p_c(D) = D$ if $fx(D)$ is a solution of c, and $p_c(D) \Leftrightarrow$ *false* if $fx(D)$ is not a solution of c. A propagator is *domain consistent* if it removes all values from the domain that cannot be extended to a valuation that satisfies c. Formally, we say propagator p_c for c over variables $\{x_1, \ldots, x_n\}$ is domain consistent if $(x_i \in S_i) \in p_c(D)$ iff $S_i = \{\theta(x_i) | \theta \in solns(c), \wedge_{i=1}^{n} \theta(x_i) \in D(x_i)\}$.

A propagation solver, denoted by solv, repeatedly applies propagators for each constraint in C until each propagator is at a fixpoint. Hence, $\mathsf{solv}(C, D) = D'$ where D' is the weakest domain such that $D' \Rightarrow D$ and $p_c(D') = D', \forall c \in C$. Note that if $D' = \mathsf{solv}(C, D)$, then $D' \Rightarrow D$ and $C \wedge D \Leftrightarrow C \wedge D'$. We assume the solver detects unsatisfiability if D' is a false domain. If the solver returns a domain D' where all variables are fixed ($\mathit{fixed}(D) = \mathit{vars}(D)$), then the solver has detected satisfiability of the problem and $\mathit{fx}(D)$ is a solution (since we assume that each propagator is checking).

Given an initial constraint problem $P_0 \equiv (C_{init}, D_{init})$, constraint programming solves P_0 by a search process that creates a tree of problems. For each problem $P \equiv (C, D)$ appearing in the tree, the search first uses the constraint solver to determine whether P can immediately be classified as satisfiable or unsatisfiable. If this is not possible, the search splits P into n subproblems as follows. Let $\mathsf{split}(C, D) = \{c_1, \ldots, c_n\}$ where $C \wedge D \Rightarrow (c_1 \vee c_2 \vee \ldots \vee c_n))$. The n children of P are the subproblems $(C \cup \{c_i\}, D)$, which are then explored by the search according to some particular order.

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_0 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 $P_i \equiv (C, \mathsf{join}(x, s, D))$, where $\mathsf{join}(x, s, D)$ modifies the domain of x to be a subset of s as follows: $\mathsf{join}(x, s, D) = (D - \{x \in D(x)\}) \cup \{x \in s \cap D(x)\}$. While this is not a strong restriction, it does rule out some kinds of constraint programming search. The main advantage of this restriction is that the set C of constraints in each subproblem is identical (and, thus, identical to the original C_{init} in P_0). As we will see later, this is crucial for our approach.

3 Subproblem Dominance and Subproblem Equivalence

Subproblem dominance is a general notion of re-usability which allows us to reuse information such as solutions, failures, and bounds on the optimization function, from one subproblem for another. Intuitively, dominance arises if the search finds a subproblem that poses the same or more constraints on the unfixed variables as a previously encountered subproblem. Formally, we define subproblem dominance and subproblem equivalence as follows:

Definition 1. *Let $P \equiv (C, D)$ and $P' \equiv (C, D')$ be two different subproblems arising during the search from some initial constraint problem. Let $F = \mathit{fixed}(D)$, $U = \mathit{vars}(C) - F$, $F' = \mathit{fixed}(D')$, and $U' = \mathit{vars}(C) - F'$. We say that P* dominates *P' iff*

- *$F = F'$ (which is equivalent to saying $U = U'$)*
- *$\exists_F.(C \wedge D') \Rightarrow \exists_F.(C \wedge D)$*

The formula $\exists_F.(C \wedge D)$ is the projecting out of the fixed variables F from the subproblem, or equivalently the projecting *onto* the unfixed variables U.

P dominates P' if the same set of variables are fixed in each, and the projection of P' onto U entails the projection of P onto U, that is P' constrains the unfixed variables more than P. Note that dominance may hold even if the fixed variables in P and P' are fixed to different values. If P dominates P', then each solution of P' corresponds to a solution of P, as formalized by the following proposition.

Proposition 1. *Suppose subproblem P dominates subproblem P'. Then $\forall \theta \in solns(P'), (\theta_U \cup fx(P)_F) \in solns(P)$.* □

The above proposition states that whenever P dominates P', every solution θ of P' corresponds to a solution of P where the literals for the unfixed variables are taken from θ (by projecting it over U), and the literals for the fixed variables are taken from the valuation $fx(P)$ projected over F.

Example 1. Consider $P_0 \equiv (C, D_0)$ where $C \equiv \{x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20\}$, $D_0 \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$. Consider two subproblems $P \equiv (C, D)$ and $P' \equiv (C, D')$ where $D_F \equiv \{x_1 = 3, x_2 = 1\}$, and $D'_F \equiv \{x_1 = 1, x_2 = 3\}$. Then $U \equiv \{x_3, x_4, x_5\}$ and $D_U \equiv D'_U \equiv \{x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$. P dominates P' since $\exists_F.(C \wedge D') \Leftrightarrow (x_3 + x_4 + 2x_5 \leq 13 \wedge D'_U) \Rightarrow (x_3 + x_4 + 2x_5 \leq 15 \wedge D_U) \Leftrightarrow \exists_F.(C \wedge D)$. □

For the remainder of the paper we will often make the implicit assumption that for the subproblem of interest P, we have $P \equiv (C, D)$ where C is the set of constraints common to all subproblems considered (i.e., the initial constraint problem $P_0 \equiv (C, D_{init})$), $D \Rightarrow D_{init}$ and that, for the given domain D, $F = fixed(D)$ and $U = vars(C) - F$.

Subproblem equivalence arises when dominance holds in both directions.

Definition 2. *Subproblems P and P' are* equivalent *iff P dominates P' and P' dominates P.*

Example 2. Consider problem $P_0 \equiv (C, D_0)$ where $C \equiv \{alldiff([x_1, x_2, x_3, x_4, x_5])\}$ and $D_0 \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$. Consider two subproblems $P \equiv (C, D)$ and $P' \equiv (C, D')$ where $D_F \equiv \{x_1 = 1, x_2 = 2\}$, and $D'_F \equiv \{x_1 = 2, x_2 = 1\}$. Both subproblems project to $alldiff([1, 2, x_3, x_4, x_4]) \wedge \{x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$ and are therefore equivalent. □

Proposition 2. *Suppose subproblems P and P' are equivalent. Then $\theta \in solns(P)$ iff $(\theta_U \cup fx(P')_F \in solns(P')$.* □

The above proposition states that whenever P and P' are equivalent, there is a bijection that maps every solution θ of P' to a solution of P where the literals for the unfixed variables are taken from θ (by projecting it over U), and the literals for the fixed variables are taken from the valuation $fx(P)$ projected over F.

3.1 Satisfaction with Caching

Detecting subproblem domination allows the search to avoid exploring the dominated subproblem and, instead, reuse the solutions of the dominating subproblem (Proposition 1). This is particularly easy when we are only interested in finding whether a problem is satisfiable, since we know the dominated subproblem

```
   cache_search($C, D$)
1     $D' :=$ solv($C, D$)
2     if $(D' \Leftrightarrow \mathit{false})$ return $\mathit{false}$
3     if $(\exists P \in \mathit{Cache}$ where $P$ dominates $(C, D'))$ return $\mathit{false}$
4     if $\mathit{fixed}(D') \equiv \mathit{vars}(D)$
5        return $\mathit{true}$
6     foreach $(x \in s) \in$ split($C, D$)
7        $S :=$ cache_search($C$, join($x, s, D$))
8        if $(S)$ return $\mathit{true}$
9     $\mathit{Cache} := \mathit{Cache} \cup \{(C, D')\}$
10    return $\mathit{false}$
```

Fig. 1. Computing satisfiability under subproblem dominance.

must have no solutions. An algorithm for satisfaction search using domination is shown in Figure 1. It is called with the cache initially empty ($\mathit{Cache} = \emptyset$). At each search node, the algorithm first 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.

3.2 Optimization with Caching

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 given by some expression e, we need only a slight modification. We assume there exists an upper bound u on the objective function so that we can have a pseudo-constraint $e \leq u$ in the problem from the beginning. Whenever a new solution is found we replace this pseudo-constraint by a tighter one reflecting the new bound. Hence, we simply replace the line 5 in Figure 1 by the following lines.

```
globally store $fx(D)$ as the current best solution
globally replace the objective function constraint by $e \leq fx(D)(e) - 1$
return $\mathit{false}$
```

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

4 Manual Keys for Caching

The principal difficulty in implementing the cache_search algorithm of Figure 1 is implementing the lookup and test for subproblem dominance (line 3 in Figure 1). As previously mentioned, a common way of doing this is by representing each subproblem with a *key* that allows dominance to be detected efficiently (preferably in $O(1)$ time). Naively, one may think that D could be used as a key to characterise subproblem $P \equiv (C, D)$. However, while D certainly uniquely identifies the subproblem, it is useless as a key since D is different for each subproblem (i.e., node) in the search tree and will never produce a match. Thus we need to find a more powerful key; one that can be identical even for equivalent subproblems with different domain constraints. We now present hand crafted keys for three problems.

4.1 Minimization of Open Stacks

The Minimization of Open Stacks Problem (MOSP) [5] can be described as follows. A factory manufactures a number of different products in batches, i.e., all copies of a given product need to be finished before a different product is manufactured. Each customer of the factory places an order requiring one or more different products. Once one product in a customer's order starts being manufactured, a stack is opened for that customer to store all products in the order. Once all the products for a particular customer have been manufactured, the order can be sent and the stack is freed for use by another order. The aim is to determine the sequence in which the products should be manufactured to minimize the maximum number of open stacks, i.e., the maximum number of customers whose orders are simultaneously active.

Most MOSP models focus on the products and directly search for the order in which each product is manufactured. Instead, the approach discussed in [4], focuses on the customers and searches for the order in which each customer's stack is closed (which in turn determines the order in which each product needs to be manufactured). The latter approach provides significant advantages [4], mainly due to the increased amount of subproblem dominance it achieves. A model for this approach using the modelling language MiniZinc [14] is as follows:

```
int: n;                         % number of customers

% which pairs of customers share at least 1 product
array[1..n,1..n] of 0..1: W;

array[1..n] of var 1..n: s;   % customer start time
array[1..n] of var 1..n: e;   % customer end order
array[1..n] of var 1..n: x;   % closing schedule, labeling vars
var 0..n: objective;          % number of stacks

constraint inverse(e,x);
constraint forall (i in 1..n) (
        minimum(s[i], [e[j] | j in 1..n where W[i,j]]));
constraint forall (t in 1..n) (sum (i in 1..n) (
        bool2int((s[i] <= t) /\ (e[i] >= t))) <= objective);

solve :: int_search(x, input_order, indomain_min, complete)
         minimize objective;
```

Each MOSP instance is defined by n, the number of customers, and a Boolean function $W(i,j)$ which tells us whether customers i and j share a product. The model uses variables $s[i] \in \{1,\ldots,n\}$ representing when each customer's stack is opened (e.g., $s[i] = j$ means the stack for customer c_i is opened at time j), and variables $e[i] \in \{1,\ldots,n\}$ representing the order in which each customer's stack is closed (e.g., $e[i] = j$ means the stack for customer c_i is the jth stack to be closed). Note that two different variables $s[i]$ and $s[j]$ can have the same value (if their customer's stacks are opened at the same time), while two different variables $e[i]$ and $e[j]$ cannot, since they represent an order and, thus, if their customer's stacks are closed at the same time, we will arbitrarily put the closure of one in front of the other. To obtain the closing schedule, the model uses a new array of variables $x[i]$ (e.g., $x[i] = j$ means the ith stack to be closed is that of customer c_j. The variables in x and e are channeled together via *inverse(e,x)* constraint, which constrains $e[i] = j$ iff $x[j] = i$. Thus it forces all variables in e (and similarly in x) to have different values. Each customer's stack must be opened before the stack of any of the customers with which they share a product closes, because the product they share must have begun production. This gives rise to the *minimum* constraints. Lastly, the number of stacks which are open during each time period is less than or equal to the total number of stacks we need, giving rise to the linear inequality constraints. Note that *bool2int* converts a Boolean to a 0-1 integer.

We label the $x[i]$ in order, i.e. we pick which customer closes first, then which customer closes second, etc. which is described by the MiniZinc `int_search` annotation.

This problem contains many subproblem equivalences that, if exploited, can yield speedups of several orders of magnitude [5]. The subproblem equivalences arise as follows: after labeling the $x[1],\ldots,x[m]$, the remaining subproblem does not depend on what order those customer's stacks were closed, but only on the *set* of customers whose stacks were closed.

Example 3. Let us consider a problem where $n = 6$, customers 1, 3 and 4 want product 1, customers 2 and 6 want product 2, and customers 3 and 5 want product 3. This is represented by the data file:

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

where a 1 in position (i,j) represents that customers i and j have a product in common. Consider the following two subproblems $P \equiv (C,D)$ where $D \equiv \{x[1] = 4, x[2] = 3, x[3] = 5\}$, and $P' \equiv (C,D')$ where $D' \equiv \{x[1] = 5, x[2] = 4, x[3] = 3\}$. If the partial assignment D does not violate the constraints and all extensions of this partial assignment fail, then any extension of $\{x[1], x[2], x[3]\} = \{3,4,5\}$ for any permutation of the assignments will also fail. $\square$

This subproblem equivalence can be exploited with a simple key: $(Vars, Vals)$ where $Vars$ is the set of x variables already labeled and $Vals$ the set of their values. In the example above, this key would be $(\{x[1], x[2], x[3]\}, \{3, 4, 5\})$. Or even more simply, we could just use $\{3, 4, 5\}$, since the order of fixing variables is fixed by the search, and the size of the set determines how many are fixed. Clearly the key is dependent on the search strategy, and this is always the case with caching constraint programs.

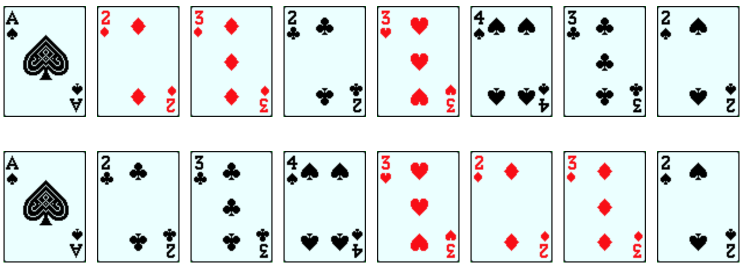


Fig. 2. Two starting sequences for the Blackhole problem that lead to an identical remaining subproblem.

4.2 Blackhole

Consider the Blackhole Problem [20], which seeks to find a solution to the Blackhole patience game. In this game the 52 playing cards (of a standard 52-card deck) 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 a pile can be played into the blackhole if it is ± 1 from the card that was played previously, with king wrapping back around to ace. The aim is to play all 52 cards. This was one of two examples used to illustrate CP with caching in [20].

Suppose the first k cards have been played. The remaining subproblem only depends on the set of unplayed cards, and the value of the last card played, since completing the game only depends on the last card played and the state of the 17 piles (which is given by the initial configuration and the unplayed cards). Subproblem equivalences arise because, for any particular assignment, there may be many permutations of the first $k-1$ assignments which also satisfy the constraints, and all of these will lead to an equivalent subproblem. Figure 2 shows an example of two assignments that lead to the same subproblem. Note that the order of the cards is completely different and this is impossible or very difficult to capture as conditional symmetries. We can exploit the subproblem equivalence by defining the key as (l, S), where l is the value of the last card played, and S is the set of unplayed cards. This key allows us to detect the equivalence of all of these subproblems and avoid the redundant search.

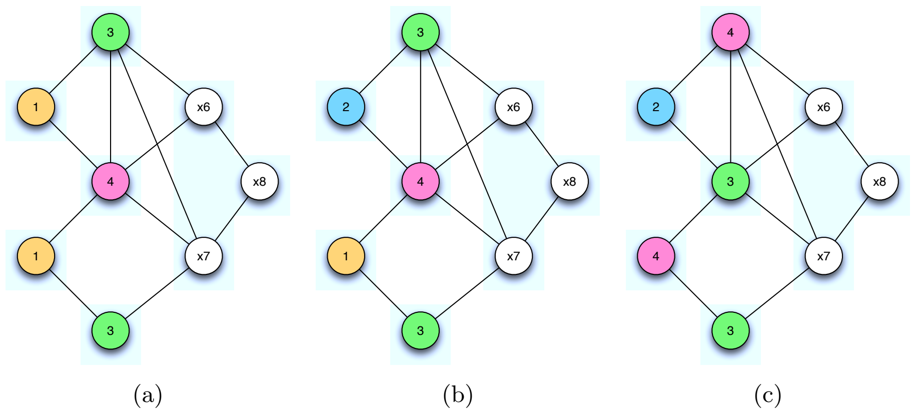


Fig. 3. Three examples of partially colored graphs

4.3 Graph Colouring

Consider the Graph Colouring Problem, which seeks to colour a graph with some set of colors (represented by integers) so that no two adjacent nodes have the same color. Given a partial (coloring) labelling of the nodes, define the *frontier* as the set of labelled nodes that are adjacent to an unlabeled node. The problem of coloring the remaining uncolored nodes is characterised by the set of labelled nodes, and the values of the labelled nodes on the frontier. It does not depend on the values of the non-frontier labelled nodes, as their value can have no further effect on the unassigned nodes.

Subproblem equivalences arise because there could be many different ways to label the non-frontier labelled nodes while producing the same frontier, and all of these lead to equivalent subproblems. Whether this occurs frequently depends on the particular structure of the graph and the labelling strategy. We can define the key as $(L, [(x, d_x)|x \in F])$, where L is the set of labelled nodes, F is the set of frontier nodes, and d_x is the value assigned to node x. This key allows us to detect the equivalence of these subproblems and avoid the redundant search. Consider the partially colored graph in Figure 3(a) which has a frontier (the second column) labelled 3, 4, 3. This frontier is identical to that of the graph in Figure 3(b). (We will refer to Figure 3(c) in Section 7.3)

5 Automatic Keys for Caching

Manually generating keys is reasonably straightforward for problems that are simple and highly structured. However, it can be very difficult for more complex problems. In this section, we examine how keys can be automatically constructed for arbitrary constraint problems.

By definition, if P dominates P' then the projection of P' onto U entails the projection of P onto U. If we could automatically construct keys that characterise the projections of subproblems, we can use such keys for automatic dominance detection. We can do this by exploiting the fact that the piecewise dominance of the projections of all constraints in C is a sufficient condition for subproblem dominance. This is formalised in the following theorem:

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

Proof.

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

The third and fifth (marked) equivalences hold because all variables being projected out in every $c \wedge D'_F$ and $c \wedge D_F$ were already fixed. □

The importance of Theorem 1 lies in the fact that it allows us to treat the projection of each constraint in C separately. Thus, if we can characterise the projection of each constraint in C, we can concatenate these, together with D_U and F, to form a projection key which completely characterises the subproblem.

5.1 Projection Keys

Naively, for any subproblem $P \equiv (C, D)$, we can define a function $key(c, D) \equiv \exists_F.(c \wedge D_F)$, where as usual $F = fixed(D)$ and $U = vars(D) - F$, to characterize the projection of the constraint c onto the unfixed variables U. There are two main problems with this. Firstly, it means that we need one key per constraint, which could make the key for the whole subproblem very large (as we will see later, more sophisticated definitions do not need to explicitly represent all constraints). And secondly, the projection of complex constraints isn't always easy to describe or represent, thus $\exists_F.(c \wedge D_F)$ may have no reasonably sized representation. We therefore use a more complex definition of $key(c, D_F)$ that allows for more flexibility and several important optimizations.

Definition 3. *Let $w(c, D_F)$ be the weakest domain on U that the propagator p_c for constraint c can return given D_F, i.e. for any D' s.t. $D' \Rightarrow D_F$ and $D' \Rightarrow D_{init}$ we have $p_c(D') \Rightarrow w(c, D_F)$.*

For monotonic propagators p_c, where $p_c(D) \Rightarrow p_c(D')$ whenever $D \Rightarrow D'$, we can compute the weakest domain $w(c, D)$ as follows. Let $D' = p_c(D_F \cup \{x \in s \mid (x \in s) \in D_{init} \wedge x \in vars(C) - F\})$, that is, the result of applying the propagator p_c to the weakest domain with fixed subset D_F. Then $w(c, D_F) = D'_U$.

Definition 4. *The* projection key $key(c, D_F)$ *of a constraint c for domain D is a set of constraints over* $vars(c)$*, such that*

$$\exists_F.(c \wedge key(c, D_F)) \wedge w(c, D_F) \Leftrightarrow \exists_F.(c \wedge D_F).$$

In general, there are many ways to correctly define $key(c, D_F)$. However, we are interested in defining $key(c, D_F)$ such that we can detect as much dominance as possible. Note that whether a particular definition of $key(c, D_F)$ is correct or not depends on the consistency level of the propagator used for constraint c. Whether a particular key is effective or not, depends on how good it is at detecting dominance and, in general, one can say that the weaker the set of constraints in the key and the more independent of D_F, the more dominance it can detect. There are two trivially correct definitions for $key(c, D_F)$: $D_{F \cap vars(c)}$ and $\exists_F.(c \wedge D_F)$. The former is correct and can always be represented, but it does not allow us to detect any dominance. The latter is correct and allows us to detect the maximum amount of dominance, but it cannot always be represented in a reasonable size.

Example 4. Let us consider four different definitions of $key(c, D_F)$ for the *alldiff* constraint and their associated dominance detection properties. Consider the subproblem $P_0 \equiv (C, D_{init})$, where $C \equiv \{alldiff([x_1, x_2, x_3, x_4, x_5, x_6])\}$ and $D_{init}(x_i) = [1..6]$ for $1 \leq i \leq 6$. Consider a subproblem $P \equiv (C, D)$ of P_0 where $D_F \equiv \{x_1 = 3, x_2 = 4, x_3 = 5\}$. Define $key_1(c, D_F) \equiv D_{F \cap vars(c)} \equiv x_1 = 3 \wedge x_2 = 4 \wedge x_3 = 5$. Define $key_2(c, D_F) \equiv \wedge_{x \in (F \cap vars(c))} x \in \{v | x_i = v, x_i \in F\} \equiv x_1 \in \{3,4,5\} \wedge x_2 \in \{3,4,5\} \wedge x_3 \in \{3,4,5\}$. Define $key_3(c, D_F) \equiv \exists_F.(c \wedge D_F) \equiv alldiff([3, 4, 5, x_4, x_5, x_6])$. Define $key_4(c, D) \equiv \wedge_{x \in vars(c) - F} x \notin \{v | x_i = v \in D_{vars(c) \cap F}\} \equiv x_4 \notin \{3,4,5\} \wedge x_5 \notin \{3,4,5\} \wedge x_6 \notin \{3,4,5\}$. While key_1 and key_3 are trivially correct, key_2 is correct because as long as the fixed vars take the same set of values, the constraints on the remaining unfixed vars will be the same, and key_4 is correct because it captures the entire effect of D_F on the unfixed variables U.

Let us now consider a different subproblem $P' \equiv (C, D')$ where $D'_F \equiv x_1 = 4 \wedge x_2 = 5 \wedge x_3 = 3$. Clearly, $key_1(c, D'_F) \equiv x_1 = 4 \wedge x_2 = 5 \wedge x_3 = 3$ does not match $key_1(c, D_F)$, while $key_2(c, D'_F) \equiv \{x_1 \in \{3,4,5\} \wedge x_2 \in \{3,4,5\} \wedge x_3 \in \{3,4,5\}\}$ matches $key_2(c, D_F)$ and shows that the *alldiff* projects to the same constraint in both subproblems. Also, while $key_3(c, D'_F) \equiv alldiff([4, 5, 3, x_4, x_5, x_6])$ does not syntactically match $key_3(c, D_F)$, it is semantically equivalent to it. Finally, $key_4(c, D'_F) \equiv x_4 \notin \{3,4,5\} \wedge x_5 \notin \{3,4,5\} \wedge x_6 \notin \{3,4,5\}$ also matches. □

We now illustrate how to use projection keys for checking subproblem dominance. Note that dominance checking, and thus projection key calculation, is only ever performed when the solver is at propagation fixpoint.

Theorem 2. *Let* $P \equiv (C, D)$ *and* $P' \equiv (C, D')$ *be subproblems arising during search from problem* $P_0 \equiv (C, D_{init})$*. Let* $F = fixed(D)$ *and* $U = vars(C) - F$*. If* $fixed(D') = fixed(D)$*, and* $\forall c \in C, key(c, D'_F) \Rightarrow key(c, D_F)$*, and* $D'_U \Rightarrow D_U$*, then* P *dominates* P'*.*

Proof.

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

The second and eighth (marked) equivalences hold because, again, all variables being projected out in each $c \wedge D'_F$ and $c \wedge D_F$ were already fixed. The fourth and sixth equivalences hold because when we are at propagation fixpoint, $D_U \Rightarrow w(c, D_F)$ for any $c \in C$. □

Definition 5. *The* subproblem projection key $skey(C, D)$ *for subproblem* (C, D) *is defined as the triple* $(F, [key(c, D_F)|c \in C], D_U)$.

We extend the implication operator $\Rightarrow$ *w.r.t. subproblem projection keys as follows:* $skey(C, D) \Rightarrow skey(C, D')$ *iff* $F' = F$ *and* $\forall c \in C, key(c, D'_F) \Rightarrow key(c, D_F)$*, and* $D'_U \Rightarrow D_U$.

Similarly we extend the equivalence operator so that $skey(C, D) \Leftrightarrow skey(C, D')$ *iff* $skey(C, D) \Rightarrow skey(C, D')$ *and* $skey(C, D') \Rightarrow skey(C, D)$.

Thanks to Theorem 2, it is obvious that if $skey(C, D') \Rightarrow skey(C, D)$, then P dominates P'. Clearly, if $skey(C, D') \Leftrightarrow skey(C, D)$, then P and P' are equivalent.

Example 5. 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 $U = \{x_4, x_5, x_6\}$. A correct projection key for the subproblem is given by the following $skey(C, D)$ $(\{x_1, x_2, x_3\}, [\{x_1, x_2, x_3\} = \{3, 4, 5\}, x_4 + 2x_5 \leq 4], \{x_4 \in \{0, 1, 2\}, x_5 \in \{0, 1, 2\}, x_6 \in \{1, 2, 6\}\})$. We choose $\exists_F(c \wedge D_F)$ to define the key for the linear constraint $c \equiv x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20$ in the projection key. □

Example 6. Consider $P \equiv (C, D)$ of Example 5 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 keys for 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 $key(P)$: $x_4 + 2x_5 \leq 3 \Rightarrow x_4 + 2x_5 \leq 4$. Similarly, $D'_U \Rightarrow D_U$. Hence, P dominates P'. □

5.2 Special Cases

The more complex definition of $key(c, D_F)$, introduced above, allows us to define $key(c, D_F) \equiv true$ in the following two cases. First, suppose $F \cap vars(c) = \emptyset$. Then $\exists_F.(c \wedge D_F) \Leftrightarrow c$ and, therefore, $key(c, D_F) \equiv true$ is trivially correct. This means that for any D_F, if c is inactive (none of its variables are fixed by D_F), we can define $key(c, D_F) \equiv true$. And second, suppose $w(c, D_F) \Rightarrow \exists_F.(c \wedge D_F)$. Then $key(c, D_F) \equiv true$ is trivially correct. This means that for any D_F, if c's propagator is strong enough that the propagation from D_F makes c *satisfied*, we can define $key(c, D_F) \equiv true$. The vast majority of the constraints satisfied by D fall under this category.

Example 7. Consider the problem $C \equiv \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_4\}$ and domain $D \equiv \{x_1 = 3, x_2 = 4, x_3 \in \{0,1,2,3,5,6\}, x_4 \in \{0,1,2,3,4,5,6\}\}$. Then, $F = \{x_1, x_2\}$ and $U = \{x_3, x_4\}$. A correct projection key for the subproblem is given by: $skey(C, D) \equiv (\{x_1, x_2\}, [true, true, true], \{x_3 \in \{0,1,2,3,5,6\}, x_4 \in \{0,1,2,3,4,5,6\}\})$. The projection key for $x_1 \neq x_2$ is *true* since it is satisfied, similarly the projection key for $x_2 \neq x_3$ is *true* since it is satisfied (even though x_3 is not fixed), and the projection key for $x_3 \neq x_4$ is *true* since none of its variables are fixed. □

Defining $key(c, D_F)$ as *true* is good because, as mentioned before, the weaker the set of constraints in the key is, and the more independent it is of D_F, the better a key it is at detecting dominance. Thus, when applicable, *true* is the best possible key for dominance detection. Further, it allows us to save space since (as we will see in the next section) it does not contribute to our key representation.

5.3 Representing and Using Projection Keys

In this section, we discuss how to represent projection keys and how to use them to cache subproblems and to carry out dominance checks. Recall that each subproblem $P \equiv (C, D)$ is characterised by $skey(C, D) \equiv (F, [key(c, D_F) | c \in C], D_U)$. We are free to take any low level representation $rep(C, D)$ of $skey(C, D)$, as long as it is uniquely identified. In particular, we can take advantage of globally known information like the constraint type of each c, their arguments, any constants involved, F, and D_{init}, to dramatically reduce the size of the representation.

We define representations $rep(c, D_F)$ for each $key(c, D_F)$ on a per constraint basis. We require that there be a bijection between tuple $(F, c, rep(c, D_F))$ and the possible values of $key(c, D_F)$. Note that different values of $key(c, D_F)$ can map to the same value of $rep(c, D_F)$, as long as F and c are sufficient to distinguish them. See Section 6 for many examples of $key(c, D_F)$ and $rep(c, D_F)$.

We also need to define a representation $rep(D_v)$ for each D_v where $v \in U$. In practice, we simply use the existing solver representation of variable domains, e.g. a bit string, a set of ranges, etc. Finally, we need to define a representation $rep(F)$ to represent F, for which we use a bit string.

We can define the representation of a subproblem (C, D), $rep(C, D)$, in one of two ways:

– The *dense representation* always includes a representation for each constraint appearing in C, and for every unfixed variable U appearing in the problem: $rep_d(C, D) \equiv (rep(F), [rep(c, D_F) | c \in C], [rep(D_v) | v \in U])$; and

– the *sparse representation* which omits constraints whose key is *true*, and omits unfixed variables whose domain is their initial domain: $rep_s(C, D) = (rep(F), [(id(c), rep(c, D_F))|c \in C, key(c, D_F) \not\equiv true], [(id(v), rep(D(v))|v \in U, D(v) \neq D_{init}(v)])$, where $id(v)$ and $id(c)$ are unique id's for variables and constraints respectively.

Clearly, both representations uniquely identify $(F, [key(c, D_F)|c \in C], D_U)$. For problems with lots of subproblem equivalences, it is generally the case that $D(v) = D_{init}(v)$ and $key(c, D_F) \equiv true$ for a very substantial portion of the variables and constraints. Thus, in the implementation we use $rep_s(C, D)$ which can take advantage of the sparsity. Note that this means that we only need to define $rep(c, D_F)$ for D_F where $key(c, D_F) \not\equiv true$.

Clearly, if $rep(C, D) = rep(C, D')$, then $skey(C, D) \Leftrightarrow skey(C, D')$ and P and P' are equivalent. If all we wanted to do was to detect subproblem equivalence, we can accomplish this very efficiently by using a hashtable to check for $rep(C, D) = rep(C, D')$, where the projection key is used as both the hashtable key and value. We can thus alter the algorithm in Figure 1 by replacing line 3 by:

if $((rep(C, D') \mapsto rep(C, D')) \in Hashtable)$ **return** *false*

and replacing line 9 by:

$Hashtable := Hashtable \cup (rep(C, D') \mapsto rep(C, D'))$

where we consider a hashtable is a set of pairs *key* $\mapsto$ *item* mapping from keys to stored items.

Example 8. Consider the problem $P \equiv (C, D)$ of Example 5. We can store its projection key $\{x_1, x_2, x_3\} = \{3, 4, 5\} \wedge x_4 + 2x_5 \leq 4 \wedge D_U$ 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 4 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 is $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 4, \ \{0, 1, 2\}, \{0, 1, 2\}, \{1, 2, 6\})$ using the dense representation, and $(\{x_1, x_2, x_3\}, (c_1, \{3, 4, 5\}), (c_2, 4), (x_4, \{0, 1, 2\}), (x_5, \{0, 1, 2\}), (x_6, \{1, 2, 6\}))$ using the sparse representation. □

To exploit subproblem dominances, and not simply equivalences, we need a method to detect when $skey(C, D') \Rightarrow skey(C, D)$. Some constraints naturally produce projections which can dominate one another, while others do not. Consider a linear constraint $x_1 + x_2 + x_3 \leq 5$. If $x_1 = 1$, this projects to $x_2 + x_3 \leq 4$. If $x_1 = 2$, this projects to $x_2 + x_3 \leq 3$, which is clearly a strictly stronger constraint. Thus, linear constraints tend to produce dominating projections. This is not the case for other kinds of constraints such as *alldiff*. No projection of the *alldiff* constraint ever strictly dominates another. e.g. *alldiff*$([1, 2, x_3, x_4, x_5])$ does not dominate and is not dominated by *alldiff*$([1, 2, 3, x_4, x_5])$. Thus, we can partition constraints into two groups: those which can generate (strictly) dominating projections, and those which cannot. Note that domain constraints D_v fall in the first group as they can strictly dominate another D'_v.

We can detect subproblem dominances as follows. We first divide the subproblem representation into two parts. The *equivalence part* $rep_e(C, D)$ consists of $rep(F)$ and $rep(c, D_F)$, for each $c \in C$ that cannot generate dominating projections. The *dominance part* $rep_d(C, D)$ consists of $rep(c, D_F)$, for each $c \in C$ that can generate dominating projections, together with $rep(D_U)$.

For dominance detection, we use only the equivalence part $rep_e(c, D_F)$ as the hashtable key, and the whole representation $rep(C, D)$ as the value. Now, looking up matches in the hashtable returns a set of subproblems with the same $rep_e(C, D)$, but possibly different $rep_d(C, D)$. We then simply compare the dominance parts of each to see if P is dominated by one of these. We alter the algorithm in Figure 1 as follows. Line 3 is replaced by:

```
if $(\exists(rep_e(C, D) \mapsto rep(C, D)) \in Hashtable$ s.t. $rep_e(C, D) = rep_e(C, D')$
    and $rep_d(C, D)$ dominates $rep_d(C, D'))$ return false
```

and line 9 is replaced by:

$$Hashtable := Hashtable \cup (rep_e(C, D') \mapsto rep(C, D'))$$

In the above, we put all the $rep(c, D_F)$ and $rep(D_v)$ that can generate dominating projections into the dominance part. However, we can actually decide on an individual basis whether to put them in the equivalence or dominance part, as long as we are consistent throughout the search. There is a trade off between speed and dominance detection strength. The more things we include in the equivalence part, the fewer matches we get and the quicker the dominance check, but the less dominance we can detect.

Example 9. Consider $P \equiv (C, D)$ of Example 5. Entailment for *alldiff* is simply identity on the set of values, while for the linear constraint we just compare fixed values. For the problem P' of Example 6 we determine the key $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 3, \{0, 1, 2\}, \{0, 1\}, \{1, 2, 6\})$. We can hash on the first two arguments of the tuple to retrieve the key for P, and then compare 3 versus 4 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.

5.4 Branch and Bound Optimization

The presentation so far has concentrated on satisfaction problems. We show that the theory can be extended to branch and bound optimization search. Without loss of generality, let us consider a minimization problem. Branch and bound optimization typically proceeds as follows. We have an expression e which describes the objective function, e.g. $e \equiv \sum_{i=1}^{n} a_i x_i$ or $e \equiv max_{i=1}^{n} x_i$, and one of the constraints in the problem is of the form $c_{obj} \equiv e \leq k$, where k is the current upper bound. If during search, a better solution with $e = k'$ is found, the search is constrained so that only solutions better than k' can be found, i.e. c_{obj} is strengthened to $c'_{obj} \equiv e \leq k' - 1$. The search continues until the search space is exhausted, and no better solution exists.

Our previous theorems were proved based on the assumption that C is the same in P and P'. However, this is not true in branch and bound optimization, as c_{obj} is strengthened to c'_{obj}. We can easily extend Theorem 2 to the case where all constraints in C monotonically increase in strength as search progresses.

Lemma 1. *Suppose $c' \Rightarrow c$, $key(c', D'_F) \Rightarrow key(c, D_F)$, and $D'_U \Rightarrow D_U$. Then $\exists_F.(c' \wedge D'_F) \wedge D'_U \Rightarrow \exists_F.(c \wedge D_F) \wedge D_U$.*

Proof.

$$\begin{aligned} & \exists_F.(c' \wedge D'_F) \wedge D'_U \\ \Leftrightarrow\ & \exists_F.(c' \wedge key(c', D'_F)) \wedge D'_U \\ \Rightarrow\ & \exists_F.(c \wedge key(c, D_F)) \wedge D_U \\ \Leftrightarrow\ & \exists_F.(c \wedge D_F)) \wedge D_U \end{aligned}$$

□

Theorem 3. *Let $P \equiv (C, D)$ and $P' \equiv (C', D')$ be subproblems arising during search. Let $F = fixed(D)$, $U = vars(C) - F$, Suppose there is a bijective mapping $m : C \rightarrow C'$ s.t. $m(c) \Rightarrow c$, i.e. every constraint $c \in C$ has a corresponding constraint $m(c) \in C'$ which is at least as strong. If $fixed(D) = fixed(D')$, $D'_U \Rightarrow D_U$, and $\forall c \in C.\ key(m(c), D'_F) \Rightarrow key(c, D_F)$, then P dominates P'.* □

The proof is analogous to that of Theorem 2, except that we apply Lemma 1 at the appropriate place. Clearly, branch and bound optimization search is covered by Theorem 3 and we can use projection keys to detect dominance as we did for satisfaction problems.

One special case to note. If e is simply a variable, then the keys generated are always true, and the optimal subproblem value is effectively cached in the unfixed variables domain for e. In this case in order to obtain dominance reasoning (rather than equivalence) we need to exclude this domain from the equivalence part of the key, and add it to the dominance part.

5.5 Redundant constraints

Problems are often modelled with redundant constraints that enhance propagation and reduce the search space. These constraints can actually be ignored in the subproblem projection. This is formalized as follows.

Lemma 2. *Suppose we have $P \equiv (C, D)$ and $P' \equiv (C', D)$ where $C' \subseteq C$ s.t. $C \Leftrightarrow C'$. Then the projection of P and P' onto U w.r.t. any D is equivalent, i.e. $\exists_F.(C \wedge D) \Leftrightarrow \exists_F.(C' \wedge D)$.*

Proof. Since $C \Leftrightarrow C'$, $C \wedge D \Leftrightarrow C' \wedge D$, thus $\exists_F.(C \wedge D) \Leftrightarrow \exists_F.(C' \wedge D)$ for any F and D. □

Theorem 4. *Let $P \equiv (C, D)$ and $P' \equiv (C, D')$ be subproblems arising during search. Let $F = fixed(D)$ and $U = vars(C) - F$. Suppose $C' \subseteq C$ s.t. $C \Leftrightarrow C'$. If $fixed(D) = fixed(D')$, $D'_U \Rightarrow D_U$ and $\forall c \in C'.\ key(c, D'_F) \Rightarrow key(c, D_F)$, then P dominates P'.* □

The proof is analogous to that of Theorem 2, except that we apply Lemma 2 at the beginning and end. This means that instead of creating projecting keys from the full set of constraints C, it is sufficient to choose a subset $C' \subseteq C$ which fully define the problem.

6 Constructing Projection Keys

In this section we discuss how to create projection keys for many common constraints. Constructing the function $key(c, D_F)$ is a form of *constraint abduction* [12], which can be a very complex task. However, we only need to perform constraint abduction on a per constraint basis. Recall that we can always define $key(c, D_F)$ as $D_{F \cap vars(c)}$ or $\exists_F.(c \wedge D_F)$. However, we want a definition for $key(c, D_F)$ that detects as much dominance as possible and is representable in a reasonable size. First we try to see if $key(c, D_F) \equiv true$ is a valid definition, as this detects the most dominance. Recall that if $F \cap vars(c) = \emptyset$ or $w(c, D_F) \Rightarrow \exists_F.(c \wedge D_F)$, then we can always define $key(c, D_F) \equiv true$. We only need to consider more complex definitions of $key(c, D_F)$ if neither of these conditions obviously hold, i.e. we only consider c if at least a variable in $vars(c)$ is fixed and D_F is not enough to satisfy c. Next we try $key(c, D_F) \equiv \exists_F.(c \wedge D_F)$, which has the maximum potential to detect dominance, but is not always representable. Thirdly, we try to find something weaker but representable. If all else fails, we use $key(c, D_F) \equiv D_{F \cap vars(c)}$, which detects no dominance, but is always representable.

6.1 Simple Constraints

Here we give the function $key(c, D_F)$ and the representation $rep(c, D_F)$ for some simple constraints. Any assumption about the minimum consistency level of the propagator is stated as necessary.

Binary Constraints Almost all commonly used propagators for binary constraints have the property that once one variable is fixed, propagation on the other variable causes the constraint to become satisfied. For such constraints, either $F \cap vars(c) = \emptyset$ or $w(c, D_F) \Rightarrow \exists_F.(c \wedge D_F)$. Thus, we can define $key(c, D_F) \equiv true$ for any D_F which means that such constraints never need to contribute to the projection key. We give a non-exhaustive list of some of these constraints, along with a sufficient set of propagation rules for the above condition to hold in Figure 4.

Linear Let c be a linear constraint of the form $c \equiv \sum_{i=1}^{n} a_i x_i \leq a_0$. Let $S = \{i | x_i \in F\}$ and d_i be the value that x_i is fixed to in D_F, for $i \in S$. Let $a'_0 = a_0 - \sum_{i \in S} a_i d_i$. We can define $key(c, D_F) \equiv \sum_{i \notin S} a_i x_i \leq a'_0$. The variables x_i and the constants a_i are implicitly known from c, and S is implicitly known from F and $vars(c)$. Thus, the representation only needs to store the single number a'_0. We can define $rep(c, D_F) = \{a'_0\}$. The $\geq$ and $=$ case of the linear constraint are analogous.

Constraint c	Sufficient set of propagation rules
$x = y$	$x = d \leftrightarrow y = d$
$x \neq y$	$x = d \rightarrow y \neq d$, $y = d \rightarrow x \neq d$
$x \geq y$	$x = d \rightarrow y \leq d$, $y = d \rightarrow x \geq d$
$x \vee y$	$\neg x \rightarrow y$, $\neg y \rightarrow x$
$(x = d) \Rightarrow b$	$x = d \rightarrow b$, $\neg b \rightarrow x \neq d$
$(x = d) \Leftarrow b$	$x \neq d \rightarrow \neg b$, $b \rightarrow x = d$
$(x \geq d) \Rightarrow b$	$x \geq d \rightarrow b$, $\neg b \rightarrow x < d$
$(x \geq d) \Leftarrow b$	$x < d \rightarrow \neg b$, $b \rightarrow x \geq d$
$(x = d') \Rightarrow (y = d)$	$x = d' \rightarrow y = d$, $y \neq d \rightarrow x \neq d'$
$y = \mathit{bool2int}(b)$	$b \leftrightarrow y = 1$, $\neg b \leftrightarrow y = 0$
$y = \mathit{abs}(x)$	$x = d \rightarrow y = \mathit{abs}(x)$, $y = d \rightarrow x \in \{d, -d\}$

Fig. 4. Sufficient conditions on propagation to ensure no key is required for the binary constraints above. Note the *bool2int*() function converts a Boolean b to a 0-1 integer y.

Reified Linear Let c be a reified linear constraint of the form $c \equiv b \rightarrow \sum_{i=1}^{n} a_i x_i \leq a_0$, that is if b is *true* the linear constraint holds and if the linear constraint does not hold then b is *false*. Let $S = \{i | x_i \in F\}$, and d_i be the value that x_i is fixed to in D_F, for $i \in S$. Let $a'_0 = a_0 - \sum_{i \in S} a_i d_i$. There are two cases. If $b = \mathit{true}$, then we can define $\mathit{key}(c, D_F) \equiv \sum_{i \notin S} a_i x_i \leq a'_0$ and $\mathit{rep}(c, D_F) = \{a'_0, \mathit{true}\}$. If $b = \mathit{false}$ the constraint is satisfied and $\mathit{key}(c, D_F) = \mathit{true}$. Otherwise if $b \notin F$, then we can define $\mathit{key}(c, D_F) \equiv b \rightarrow \sum_{i \notin S} a_i x_i \leq a'_0$ and $\mathit{rep}(c, D_F) = \{a'_0, \mathit{false}\}$. Note that the representation is thus simply the representation of the non-reified linear, plus an additional Boolean value. Other kinds of reified constraints can be handled in a similar manner.

Clause Let c be a clausal constraint of the form $c \equiv \vee_{i=1}^{n} l_i$ where l_i are Boolean literals (either b_i or $\neg b_i$). Let $S = \{i | b_i \in F\}$. Suppose $\exists i \in S$ s.t. $l_i = \mathit{true}$ (that is $l_i \equiv b_i$ and $D(b_i) = \mathit{true}$ or $l_i \equiv \neg b_i$ and $D(b_i) = \mathit{false}$). Then $w(c, D_F) \Rightarrow \exists_F.(c \wedge D_F)$ and we can define $\mathit{key}(c, D_F) \equiv \mathit{true}$ and $\mathit{rep}(c, D_F) = \{\mathit{true}\}$. Suppose $\nexists i \in S$ s.t. $l_i = \mathit{true}$. Then we can define $\mathit{key}(c, D_F) \equiv \vee_{i=1, i \notin S}^{n} l_i$. Once again, S is implicitly known from F and $\mathit{vars}(c)$, so in fact, we don't need to store any additional information. We can define $\mathit{rep}(c, D_F) = \{\mathit{true}\}$ in the first case and $\mathit{rep}(c, D_F) = \{\mathit{false}\}$ in the second.

Arithmetic constraints Most arithmetic constraints like $c \equiv z = x \times y$, $c \equiv z = x \mod y$, do not project well. Thus, we can only use as last resort $\mathit{key}(c, D_F) \equiv D_{F \cap \mathit{var}(c)}$ and $\mathit{rep}(c, D_F) = [D_v | v \in (F \cap \mathit{var}(c))]$.

6.2 Global Constraints

Constraint programming models often make use of global constraints to define important subproblems of the model. On the face of it defining a projection key for a complex global constraint may seem a daunting task. But in many cases it is quite straightforward.

For many global constraints, we can use the following theorem to work out a correct definition of $key(c, D_F)$:

Theorem 5. *Let $c \Leftrightarrow \wedge_{i=1}^{n} c_i$ be a decomposition of c that introduces no new variables. If $w(c, D_F) \Rightarrow w(c_i, D_F)$ for all i, then we can define $key(c, D_F) \equiv \wedge_{i=1}^{n} key(c_i, D_F)$.*

Proof.

$$\begin{aligned}
&\exists_F.(c \wedge key(c, D_F)) \wedge w(c, D_F)\\
\Leftrightarrow\ &\exists_F.(\wedge_{i=1}^{n}(c_i \wedge key(c_i, D_F))) \wedge \wedge_{i=1}^{n} w(c_i, D_F) \wedge w(c, D_F)\\
\Leftrightarrow\ &\wedge_{i=1}^{n}(\exists_F.(c_i \wedge key(c_i, D_F)) \wedge w(c_i, D_F)) \wedge w(c, D_F)\\
\Leftrightarrow\ &\wedge_{i=1}^{n} \exists_F.(c_i \wedge D_F) \wedge w(c, D_F)\\
\Leftrightarrow\ &\exists_F.(\wedge_{i=1}^{n}(c_i \wedge D_F)) \wedge w(c, D_F)\\
\Leftrightarrow\ &\exists_F.(c \wedge D_F)
\end{aligned}$$

The reverse direction of the last equivalence holds because for any correct propagator, we must have $\exists_F.(c \wedge D_F) \Rightarrow w(c, D_F)$ if D is a fixpoint of p_c.

Thus, if our global constraint c can be decomposed into simpler constraints for which we already have correctly defined $key(c_i, D_F)$, we can simply take their conjunction as the definition for $key(c, D_F)$. Note that the solver does not need to use the constraint decomposition for propagation. The decomposition is introduced purely to define $key(c, D_F)$. For this reason it is more effective to take a simple, weak decomposition, than to take a complicated and strong decomposition with many redundant constraints. Simpler decompositions with fewer constraints produce fewer keys to conjoin, and there is a higher chance that our global propagator has stronger propagation than the decomposition.

Here we give the function $key(c, D_F)$ and the representation $rep(c, D_F)$ for some global constraints. Again, any assumption about the minimum consistency level of the propagator is stated as necessary.

Alldiff The *alldiff* constraint is defined by (and decomposes to):

$$c \equiv alldiff([x_1, \ldots, x_n]) \Leftrightarrow \wedge_{i=1}^{n} \wedge_{j=1, j \neq i}^{n} x_i \neq x_j.$$

If our propagator enforces at least $x_i = v \rightarrow x_j \neq v$ for all $i \neq j$, then we can apply Theorem 5 with this decomposition. Since all of the constraints in the decomposition are binary, $key(c_i, D_F) \equiv true$ for all of them. So we can define $key(c, D_F) \equiv true$ for all D_F for the *alldiff* constraint, which means we don't need to define $rep(c, D_F)$.

And Let c be a global conjunction constraint of the form $c \equiv y = \wedge_{i=1}^{n} x_i$. We can decompose c into clausal constraints: $c \Leftrightarrow \wedge_{i=1}^{n}(\neg y \vee x_i) \wedge (y \vee \vee_{i=1}^{n} \neg x_i)$. Suppose our *and* propagator is domain consistent. Then we can apply Theorem 5. So we can define $key(c, D_F) \equiv \wedge_{i=1}^{n} key(\neg y \vee x_i, D_F) \wedge key(y \vee \vee_{i=1}^{n} \neg x_i, D_F)$. Now $key(c_i, D_F) \equiv true$ for all the binary clauses in the decomposition, and so this simplifies to $key(c, D_F) \equiv key(y \vee \vee_{i=1}^{n} \neg x_i, D_F)$. Thus, the key and representation is exactly the same as that for a clause, which we have already covered in the previous subsection.

Element Let c be an element constraint of the form $c \equiv \mathit{element}(x, [a_1, \ldots, a_n], y)$ where a_i are constants. The *element* constraint is defined by (and decomposes to): $\mathit{element}(x, [a_1, \ldots, a_n], y) \Leftrightarrow \wedge_{i=1}^{n}(x = i \rightarrow y = a_i)$. Suppose our propagator enforces at least $x = i \rightarrow y = a_i$ and $y \neq a_i \rightarrow x \neq i$. Then, we can apply Theorem 5 with this decomposition. Since all of the constraints in the decomposition are binary, $\mathit{key}(c_i, D_F) \equiv \mathit{true}$ for all of them. Hence just as for *alldiff*, we don't need to define $\mathit{rep}(c, D_F)$.

Inverse The *inverse* constraint is defined by (and decomposes to):

$$c \equiv \mathit{inverse}(x_1, \ldots, x_n, y_1, \ldots, y_n) \Leftrightarrow \wedge_{i=1}^{n} \wedge_{j=1}^{n} x_i = j \leftrightarrow y_j = i.$$

If our propagator enforces at least $x_i = j \rightarrow x_j = i$ and $x_i \neq j \rightarrow x_j \neq i$, then we can apply Theorem 5 with this decomposition. Since all of the constraints in the decomposition are binary, again we don't need to define $\mathit{rep}(c, D_F)$.

Minimum Let c be a minimum constraint of the form $c \equiv y = \min_{i=1}^{n} x_i$. Let $S = \{i | x_i \in F\}$ and let d_i be the value that x_i is fixed to in D_F for $i \in S$. Let $d = \mathit{min}_{i \in S} d_i$. Assume we have a special constant ∞, whose value cannot be taken by any integer variable in the system. There are three cases. If $y \notin F$ and $S \neq \emptyset$, then we can define $\mathit{key}(c, D_F) \equiv y = \min(d, \min_{i \notin S} x_i)$ and $\mathit{rep}(c, D_F) = (\infty, d)$. If $y \in F$ and $S \neq \emptyset$ then we can define $\mathit{key}(c, D_F) \equiv e = \min(d, \min_{i \notin S} x_i)$ and $\mathit{rep}(c, D_F) = (e, d)$ where e is the fixed value of y ($y = e \in D_F$). Finally, if $y \in F$ and $S = \emptyset$, then we can define $\mathit{key}(c, D_F) \equiv e = \min_{i=1}^{n} x_i$ and $\mathit{rep}(c, D_F) = (e, \infty)$ where e is defined as before. The *maximum* constraint is analogous to *minimum*.

Global Cardinality Constraint The global cardinality constraint *gcc* constraint [17] requires that the number of variables in $\{x_1, \ldots, x_n\}$ taking value j is n_j, where here we assume n_j are fixed, is defined by:

$$c \equiv \mathit{gcc}([x_1, \ldots, x_n], [n_1, \ldots, n_k]) \Leftrightarrow \wedge_{i=1}^{k}(\sum_{j=1}^{n} \mathit{bool2int}(x_j = i) = n_i).$$

Let $S = \{i | x_i \in F\}$. Let $n'_j = n_j - \sum_{i \in S} \mathit{bool2int}(x_i = j)$ for $j = 1, \ldots, k$. Then we can define $\mathit{key}(c, D_F) \equiv \wedge_{j=1}^{k}(\sum_{i \notin S} \mathit{bool2int}(x_i = j) = n'_j)$. S is known implicitly from $\mathit{vars}(c)$ and F, so we can simply define $\mathit{rep}(c, D_F) = \{n'_1, \ldots, n'_k\}$. The *global_cardinality_low_up* constraint which requires the number of values taking value j to be between lower bound l_j and upper bound u_j,defined by

$$\mathit{gcclu}([x_1, \ldots, x_n], [l_1, \ldots, l_k], [u_1, \ldots, u_k]) \Leftrightarrow \wedge_{i=1}^{k}(l_i \leq \sum_{j=1}^{n} \mathit{bool2int}(x_j = i) \leq u_i),$$

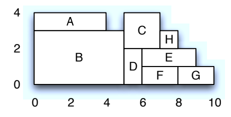
is analogous.

Cumulative Constraint The *cumulative* constraint [1] requires that n tasks with start time s_i, duration d_i and resource utilization r_i are scheduled such that at no time more than b resources are used. It can be defined as:

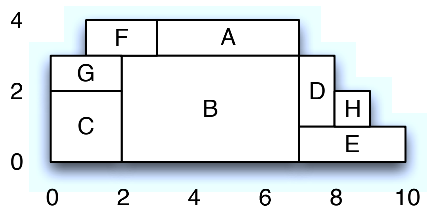
$$\begin{aligned} c &\equiv \mathit{cumulative}([s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_1, \ldots, r_n], b]) \\ &\equiv \bigwedge_t \sum_{i=1}^{n} \mathit{bool2int}((s_i \leq t) \wedge (s_i > t - d_i)) \times r_i \leq b, \end{aligned}$$

where t ranges from $\min_{i=1}^{n} \min(D_{init}(s_i))$ to $\max_{i=1}^{n} \max(D_{init}(s_i) + d_i)$ which are all the time periods when tasks can be scheduled. We consider only the version where s_i are variables and r_i, d_i and b are constants. We present a handcrafted key $\mathit{key}(c, D_F)$ which is intuitively obvious, but which isn't the most powerful possible. Ideally, we would like to define $\mathit{key}(c, D) \equiv \exists_F.(c \wedge D_F)$. However, $\exists_F.(c \wedge D_F)$ cannot really be represented in any simple way.

Intuitively, if we fix the start times of the exact same set of tasks, in such a way that the time profile of the resources they consumed is exactly identical, then, the constraint on the remaining tasks must be the same. We define the time profile of consumed resources as follows: Let $S = [i | s_i \in F]$. Let $p(t) = \sum_{i \in S} \mathit{bool2int}(D_F(s_i) \leq t \wedge D_F(s_i) > t - d_i) \times r_i$ be the amount of resources consumed by the fixed tasks at time t. $p(t)$ is a set of constants determined by D_F. Now we only have to examine the profile for times that unfixed tasks can still actually make use of. Let t_{min} be the minimum start time of an unfixed tasks in $w(c, D_F)$, and t_{max} be the maximum end time of an unfixed task in $w(c, D_F)$. Now, we define $\mathit{key}(c, D_F) \equiv \wedge_{t \in t_{min}..t_{max}} p(t) = \sum_{i \in S} \mathit{bool2int}(s_i \leq t \wedge s_i > t - d_i) \times r_i$, which is a constraint over $F \cap \mathit{vars}(c)$. Clearly, $\exists_F.(c \wedge \mathit{key}(c, D_F)) \wedge w(c, D_F) \Leftrightarrow \exists_F.(c \wedge D_F)$. Now, any two D_F and D'_F which has the same set of fixed tasks and produce the same t_{min} and t_{max} and $p(t)$ over this range of times will have matching keys.



(a)



(b)

Fig. 5. Different fixed sub-schedules for *cumulative* leading to the same key.

Example 10. Consider a cumulative constraint with fixed tasks A, B, C, D, E, F, G, H illustrated in Figure 5(a) with task durations $d = [4, 5, 2, 1, 3, 2, 2, 1]$ and resource requirements $r = [1, 3, 2, 2, 1, 1, 1, 1]$ and resource bound $b = 4$. The start times illustrated in Figure 5(a) are $s = [0, 0, 5, 5, 6, 6, 8, 7]$. Assume that no other unfixed task (not shown) can start before time 7. Then the key is represented $p(t)$

for $t = 7$ onwards, that is $p(7) = 3, p(8) = 2, p(9) = 1, p(t) = 0, t > 9$. Now consider the cumulative constraint with the same fixed tasks as illustrated in Figure 5(b), where again no other task can start before time 7. The key is identical even though the fixed schedules and even profiles of the two subproblems are different. □

6.3 More Global Constraints

For some global constraints, the best projection keys can only be expressed as constraints over the *internal variables*, I, of the global propagator. We cannot rigorously define $key(c, D_F)$ as constraints over such internal variables I unless they are explicitly defined. Such internal variables are often defined by the decompositions of the global constraint.

Suppose $c \Leftrightarrow \exists_I.(\wedge_{i=1}^n c_i)$ is a decomposition that introduces new variables I. We extend $P \equiv (C, D)$ to $P' \equiv (C', D')$ as follows: Define $D' = D \cup \{v \in D_{init}(v) | v \in I\}$. Define c' to be a constraint over $vars(c) \cup I$ s.t. $c' \Leftrightarrow \wedge_{i=1}^n c_i$. Define $C' = (C - \{c\}) \cup \{c'\}$. Then, P and P' are related as follows: $C \wedge D \Leftrightarrow \exists_I.(C' \wedge D')$. Clearly, P is satisfiable iff P' is, and there is a surjective mapping from the solutions of P' to P. Thus we can work with the problem P' instead.

Let $F' = F \cup fixed(I)$ and $U' = vars(C') - F'$. Then $D'_{U'} = D_U \cup D'_{U \cap I}$. Now we can apply Theorem 5 to P'. This tells us that if $w(c', D'_{F'}) \Rightarrow w(c_i, D'_{F'})$ for all i, then we can define $key(c', D'_{F'}) \equiv \wedge_{i=1}^n key(c_i, D'_{F'})$. Thus the projection key for P', as compared to P, changes in the following ways: F becomes $F \cup fixed(I)$, D_U becomes $D_U \cup D'_{U \cap I}$, and $key(c, D_F)$ which we didn't know how to define before, becomes $key(c', D'_{F'}) \equiv \wedge_{i=1}^n key(c_i, D'_{F'})$.

Although we had to explicitly define c' and I in order to apply Theorem 5, c' and I only have to be logically defined. They do not have to be implemented as real constraints or real variables in the system. For example, suppose the global propagator for c has some propagator-specific internal data structures, such that the domains for the variables in I are uniquely determined by them. Then variables I exist logically in the propagator and the propagator is already implementing a constraint over $vars(c) \cup I$. Also, all extra parts of the key required by the decomposition, i.e. $fixed(I)$, $D'_{U \cap I}$ and $key(c', D'_{F'}) \equiv \wedge_{i=1}^n key(c_i, D'_{F'})$, depend only on c_i and I. So if I is kept internally in the propagator, the propagator can produce all of the extra key parts we need. Thus, we can completely encapsulate the decomposition in the propagator without the rest of the system ever needing to know anything about I, or that we extended from P to P'. To express this, we can simply define $key(c, D_F) \equiv \wedge_{i=1}^n key(c_i, D'_{F'}) \wedge D'_{U \cap I}$ and $rep(c, D_F) = fixed(I) \ ++ \ rep(D'_{U \cap I}) \ ++ \ [(id(c_i), rep(c_i, D'_{F'})) | key(c_i, D'_{F'}) \not\equiv true]$ (where $++$ represents sequence concatenation), and use the normal key construction algorithm as applied to P.

6.4 Table

The *table* constraint is defined by (and decomposes to):

$$c \equiv table([x_1, \ldots, x_n], [a_{i,j} | i = 1..k, j = 1..n]) \Leftrightarrow \exists_I.(\wedge_{i=1}^k \wedge_{j=1}^n r_i \rightarrow x_j = a_{i,j} \wedge \vee_{i=1}^k r_i),$$

where $I = \{r_1, \ldots, r_k\}$ are introduced Boolean variables. Suppose our propagator internally keeps track of the domains of the r_i (which tuples are still possible) and enforces at least $r_i \rightarrow x_j = a_{i,j}$, $x_j \neq a_{i,j} \rightarrow \neg r_i$. Then we can extend the problem to include the r_i and apply Theorem 5. All of the constraints in the decomposition are binary and have $key(c_i, D_F) \equiv true$ except for the clause $\vee_{i=1}^{k} r_i$. Thus we can define $key(c, D_F) \equiv key(\vee_{i=1}^{k} r_i, D_F) \wedge D_{U \cap I}$ and $rep(c, D_F) = fixed(I) \ {+}{+} \ rep(D_{U \cap I}) \ {+}{+} \ \{rep(\vee_{i=1}^{k} r_i, D_F)\}$. Now, since the variables in I are all Boolean, $rep(D_{U \cap I})$ is empty. We represent the projection for the clause with just a single Boolean value b indicating whether it is satisfied by D_F or not. So $rep(c, D_F) = fixed(I) \ {+}{+} \ [b]$.

Cumulative Now we define a projection key for the *cumulative* constraint via decomposition. The *cumulative* constraint can be decomposed in the following way:

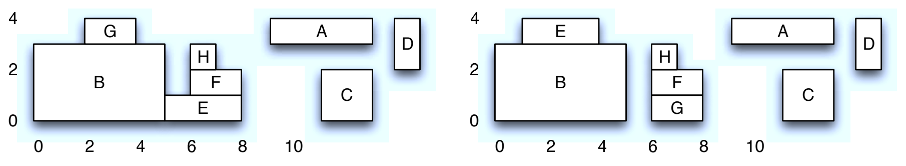
$$\begin{aligned} & cumulative([s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_1, \ldots, r_n], b]) \\ \Leftrightarrow & \bigwedge_t ((so_{i,t} \leftrightarrow s_i \leq t) \wedge (eo_{i,t} \leftrightarrow s_i > t - d_i) \wedge (o_{i,t} \leftrightarrow so_{i,t} \wedge eo_{i,t}) \wedge \\ & \sum_{i=1}^{n} bool2int(o_{i,t}) \times r_i \leq b, \end{aligned}$$

where t is appropriately limited by the ranges of s_i and d_i.

Suppose our propagator internally keeps track of the domains of the introduced variables $I = \{so_{i,t}\} \cup \{eo_{i,t}\} \cup \{o_{i,t}\}$ and enforces the constraints in the decomposition on them. Then we can extend the problem to include I and apply Theorem 5. All the reified constraints are binary and have $key(c, D_F) \equiv true$. However, the Boolean constraints and the linear constraints may be active. We know how to define keys and representations for those from Section 6.1. Thus we simply conjunct them and $D_{U \cap I}$ together to define $key(c, D_F)$ for the cumulative constraint. $rep(c, D_F)$ consists of the concatenation of $F \cap I$ along with the representations of the active constraints in the decomposition, and the representation of $D_{U \cap I}$.

Now, let us consider what this projection key means semantically. The projection key of linear constraints is $key(c, D_F) \equiv \sum_{i \notin S} a_i x_i \leq a'_0$. Since we have one linear constraint for each time t, these keys tell us exactly how much resource is available at each time t for the set of tasks which can still fit there. This is very similar to the resource profile used in the handcrafted key from Section 6.2. However, there is one major difference. It is possible for the linear constraint for some time t to become satisfied through propagation, in which case $key(c, D_F) \equiv true$ and that part of the profile no longer matters. In the handcrafted key, we manually took into account such times if they were at the beginning or at the end of the schedule. Here, the projection key does it for all such time t.

Example 11. Consider the *cumulative* example from Example 10 once more but this time where B, E, F, G, H are fixed as shown in Figure 6(a). The earliest start time for remaining tasks A, C, D is 4 (for task A), so clearly the previously



(a) (b)

Fig. 6. Different fixed sub-schedules for *cumulative* leading to the same key when using decomposition based keys.

defined key for this partial schedule is different from that of the state shown in Figure 6(b) since the profiles at time 5 are different. In the new key however, in both cases the start times for A, C and D are exactly 4.., 7.. and $\{5\} \cup 7..$. The linear constraint at $t = 4$ is satisfied since only $o_{A,4}$ is unfixed, and task A fits in the remaining 1 unit. The linear constraint at $t = 5$ is satisfied since only $o_{A,5}$ and $o_{D,5}$ are unfixed, and both A and D can simultaneously fit in the remaining 3 or 4 units in Figure 6(a) and Figure 6(b) respectively. The keys for the linear constraint for $t \geq 7$ are clearly equivalent. Hence the decomposition based key will detect these situations as equivalent with respect to *cumulative* $\square$

As can be seen, using decomposition allows us to define projection keys for even complicated constraints in a straightforward manner. Further, such keys are often fairly simple and powerful. In fact, they are often better at detecting dominance than the intuitively obvious keys that a human may design.

All global constraints supported by MiniZinc [14] have decompositions, hence, under the assumption that the decompositions propagate no stronger than the global propagator actually used (which is always true in practice) we can automatically cache all MiniZinc models.

7 Examples of Automatic Caching

In this section, we apply our algorithm to several problems to illustrate what the projection keys look like and what kinds of subproblem dominances they can detect. We also compare our automatically generated keys with handcrafted keys that a human may design. In most cases, our automatically generated projection keys contain significant amounts of redundant information. Thus, a handcrafted key will almost always be smaller in size. However, there are several advantages to automatically generating projection keys. Firstly, subproblem dominances are often very hard to identify and describe. Secondly, altering a solver to exploit subproblems dominances for a particular problem via handcrafted keys can require a tremendous amount of work. Thirdly, if the problem is ever altered, or side constraints added, etc, the handcrafted key will have to be changed, its correctness will have to be proven, and the system will have to be altered. All of this is very tedious and error prone. On the other hand, a system which implements caching via projection keys can handle all this automatically. We go through the three example problems presented previously.

7.1 Minimization of Open Stacks Problem

The model for this problem was presented in Section 1. The linear constraints $\sum_{i=1}^{n} \mathit{bool2int}((s[i] \leq t) \land (e[i] \geq t)) \leq \mathit{objective}$ are flattened into primitive constraints before being sent to the solver, since solvers typically only handle primitive constraints natively. Explicitly, those linear inequalities would be replaced by:

```
array[1..n,1..n] of var bool: so;
array[1..n,1..n] of var bool: eo;
array[1..n,1..n] of var bool: o;

constraint forall (t in 1..n) (
        forall (i in 1..n) (
               so[i,t] <-> (s[i] <= t) /\
               eo[i,t] <-> (e[i] >= t) /\
               o[i,t] <-> (so[i,t] /\ eo[i,t])
        ) /\
        sum (i in 1..n) (bool2int(o[i,t]) <= objective)
);
```

Consider the problem data used in Example 4.1 and assume we are currently solving the problem with $\mathit{objective} \leq 3$. Let us consider the following two subproblems $P \equiv (C, D)$ where $D_F \equiv \{x[1] = 4, x[2] = 3, x[3] = 5\}$, and $P' \equiv (C, D')$ where $D'_F \equiv \{x[1] = 5, x[2] = 4, x[3] = 3\}$. We know from manual analysis that these two should produce equivalent subproblems. Let us look at the automatically generated projection keys.

In P, the *inverse* constraint gives $e[4] = 1, e[3] = 2, e[5] = 3$. The *minimum* constraints give $s[1] = 1$, $s[2] \in \{4, 5, 6\}$, $s[3] = 1$, $s[4] = 1$, $s[5] = 2$, $s[6] \in \{4, 5, 6\}$. The binary reification constraints and the clausal constraints will now fix many of the *so*, *eo* and *o*. Everything with $t \leq 3$ becomes fixed. Everything related to customers $3, 4, 5$ become fixed. All the $\mathit{so}[i, t], t \geq 4$ where customer i's stack is currently open (i.e. customer 1) become fixed. All the $\mathit{eo}[i, t], t = 4$ becomes fixed. All the $o[i, 4]$ where customer i's stack is currently open become fixed. All the rest are unfixed. In D_U, $s[2] \in \{4, 5, 6\}, s[6] \in \{4, 5, 6\}, e[1], e[2], e[6] \in \{4, 5, 6\}, x[4], x[5], x[6] \in \{4, 5, 6\}$, and all other unfixed variables are at their initial domains. Now let's consider the constraint projections. The $\mathit{inverse}(e, x)$ constraint and the binary reification constraints $\mathit{so}[i, t] \leftrightarrow (s[i] \leq t)$ and $\mathit{eo}[i, t] \leftrightarrow (e[i] \geq t)$ all have $\mathit{key}(c, D_F) \equiv \mathit{true}$. Of the *minimum* constraints, the ones regarding $s[i], i = 1, 3, 4, 5$ are all satisfied and have $\mathit{key}(c, D_F) \equiv \mathit{true}$. The remaining two have no fixed variables and also have $\mathit{key}(c, D_F) \equiv \mathit{true}$. The linear constraints for $t = 1, 2, 3$ are all satisfied and have $\mathit{key}(c, D_F) \equiv \mathit{true}$. The linear constraint for $t = 4$, projects to $o[2, 4] + o[6, 4] \leq 2$, which is satisfied, so $\mathit{key}(c, D_F) \equiv \mathit{true}$. The linear constraints for $t = 5, 6$ project to $o[1, t] + o[2, t] + o[6, t] \leq 3$ which is also satisfied, and so $\mathit{key}(c, D_F) \equiv \mathit{true}$. Of the clausal constraints $o[i, t] \leftrightarrow (\mathit{so}[i, t] \land \mathit{eo}[i, t])$, all the ones with $t = 1, 2, 3$ are satisfied and have $\mathit{key}(c, D_F) \equiv \mathit{true}$. All the ones with $i = 3, 4, 5$ are satisfied and have $\mathit{key}(c, D_F) \equiv \mathit{true}$. The ones with $o[i, t], i = 2, 6, t = 4, 5, 6$ have no fixed variables and have $\mathit{key}(c, D_F) \equiv \mathit{true}$. The one with $o[1, 4]$ is satisfied and has $key(c, D_F) \equiv true$. The ones with $o[i,t], i = 1, t = 5, 6$ are active, and have $key(c, D_F) \equiv \neg eo[i,t] \vee o[i,t]$. As can be seen, almost all the constraints have $key(c, D_F) \equiv true$. This is typical of problems with lots of subproblem equivalences. Many constraints will either be inactive, or will propagate in such a way that they are no longer relevant for the remaining subproblem.

Let us now consider P'. Propagation on the *inverse* constraint gives $e[5] = 1, e[4] = 2, e[3] = 3$. The *minimum* constraints give $s[1] = 2$, $s[2] \in \{4, 5, 6\}$, $s[3] = 1$, $s[4] = 2$, $s[5] = 1$, $s[6] \in \{4, 5, 6\}$. Once again, the binary reification constraints and the clausal constraints will now fix many of the *so*, *eo* and *o* variables. We can see, from the previous description, that F, D_U and $key(c, D_F)$ only depend on the set of closed customers and on the set of currently open customers. Since these are identical for P and P', the projection keys of these two subproblems match. The key point here is that although $e[3], e[4], e[5], s[1], s[3], s[4], s[5]$ take on different values in P and P', their effect on the variables with $t \geq 4$ are identical. And since the remaining non-satisfied constraints in each subproblem only involve variables with $t \geq 4$, the subproblems end up being equivalent.

In general, if the set of closed customers is S, then the set of currently open customers is $[i | i \notin S, \exists j \in S.W(i,j)]$, which clearly only depends on S. Thus, any two subproblems with the same set of closed customers will have the same projection key. This means that our automatically generated projection key detects exactly the same amount of subproblem equivalence as the handcrafted key presented in Section 4. However, the projection key is clearly much larger since it involves the domains of all unfixed variables. On the other hand, this subproblem equivalence is fairly hard to identify manually. The MOSP was the subject of the 2005 Modelling Challenge [19], and of the 13 entrants into the competition, only 3 identified this subproblem equivalence. Thus, such subproblem equivalences are hard to identify even for expert problem modellers.

7.2 Blackhole

The Blackhole Problem can be modelled as follows:

```
array[1..17, 1..3] of int: layout;      % layout:  pile and layer
array[1..416, 1..2] of int: neighbours; % next card is +/- 1

array[1..52] of var 1..52: y;                 % Position of card
array[1..52] of var 1..52: x :: is_output;    % Card at position

constraint y[1] == 1;     % Ace of spades starts in blackhole
constraint inverse(x,y);
constraint forall (i in 1..17, j in 1..2) (
        y[layout[i,j]] < y[layout[i,j+1]]);
constraint forall (i in 1..51) (
        table([x[i], x[i+1]], neighbours));

solve :: int_search(x, input_order, indomain_min, complete)
      satisfy;
```

We take card value v to mean the $(v-1\%13)+1$th number of the $(v-3/13)+1$th suit (spades, hearts, clubs, diamonds), e.g. 1 is the ace of spades, 14 is the ace of hearts, 27 is the ace of clubs, 40 is the ace of diamonds. The array *layout* gives the layout of the cards in the 17 piles. The array *neighbours* lists pairs of cards which are ±1 w.r.t to each other, and is used in the *table* constraints to enforce the ±1 condition. For simplicity of explanation we assume that the *table* constraint's propagator is only checking, i.e. when all its variables are fixed, it checks whether that set of values is in its table and fails if it isn't. The search places cards in the blackhole in order.

Consider two subproblems $P \equiv (C, D)$ where $D_F \equiv \{x_1 = 1, x_2 = 41, x_3 = 42, x_4 = 28, x_5 = 16, x_6 = 4, x_7 = 29, x_8 = 2\}$ and $P' \equiv (C, D')$ where $D_F \equiv \{x_1 = 1, x_2 = 28, x_3 = 29, x_4 = 4, x_5 = 16, x_6 = 41, x_7 = 42, x_8 = 2\}$, illustrated in Figure 2. Assume that both partial assignments satisfy the constraints. According to our manual analysis, since the last cards are the same and the set of unplayed cards are the same, P and P' should lead to equivalent subproblems.

Let S denote the set of played cards, i.e. $\{1, 2, 4, 16, 28, 29, 41, 42\}$, which is the same for both P and P'. In P, the *inverse* constraint will enforce $x_i \neq v$ for $i \geq 9, v \in S$, and $y_i \geq 9$ for $i \notin S$. The *table* constraints perform no propagation. The inequality constraints perform some propagation. For example, if card i is under card j and neither have been played, then since $y_i > y_j$, $y_j \geq 9 \rightarrow y_i \geq 10$. Now, the *inverse* constraint and the binary inequalities all have $key(c, D_F) \equiv true$. A *table* constraint which is only checking, has $key(c, D_F) \equiv true$ if it is satisfied or inactive, and has $key(c, D_F) \equiv D_{F \cap vars(c)}$ otherwise. The *table* constraints involving only x_i where $i \leq 8$ are all satisfied and has $key(c, D_F) \equiv true$. The ones involving only x_i where $i \geq 9$ are all inactive and has $key(c, D_F) \equiv true$. Thus, the only one with a non-trivial key is the one involving x_8 and x_9, which has $key(c, D_F) \equiv x_8 = 2$. Therefore, the projection key characterizes the subproblem by F, D_U, and the last card played $x_8 = 2$. Since F and D_U are uniquely determined by the set of unplayed cards, our projection key detects the same amount of equivalence as the handcrafted key. However, it is larger in size, due to the need to store D_U.

7.3 Graph Colouring

In Section 4 we gave a handcrafted key for the graph colouring problem. It turns out that this key, although intuitively obvious, is not the strongest key possible for detecting subproblem equivalences. Projection gives a better key. Consider the simple graph colouring problem shown in Figure 3.

Consider $P \equiv (C, D)$ where $D_F \equiv \{x_1 = 1, x_2 = 1, x_3 = 3, x_4 = 4, x_5 = 3\}$ illustrated in Figure 3(a). After propagation, we have $F = \{x_1, x_2, x_3, x_4, x_5\}$ and $D_U \equiv \{x_6 \in \{1, 2\}, x_7 \in \{1, 2\}, x_8 \in \{1, 2, 3, 4\}\}$. Since all constraints are binary inequalities, we have $key(c, D_F) \equiv true$. Thus, the projection key completely characterises the subproblem by F and D_U. Consider $P' \equiv (C, D')$ where $D'_F \equiv \{x_1 = 2, x_2 = 4, x_3 = 4, x_4 = 3, x_5 = 3\}$ illustrated in Figure 3(c). After propagation, we have $F' = \{x_1, x_2, x_3, x_4, x_5\}$ and $D'_{U'} \equiv \{x_6 \in \{1, 2\}, x_7 \in \{1, 2\}, x_8 \in \{1, 2, 3, 4\}\}$. Clearly, $F = F'$ and $D_U = D'_U$ and our projection key shows that these two subproblems are equivalent. However, our handcrafted key would have characterised P with $(\{x_1, x_2, x_3, x_4, x_5\}, [(x_3, 3), (x_4, 4), (x_5, 3)])$ and P' with $(\{x_1, x_2, x_3, x_4, x_5\}, [(x_3, 4), (x_4, 3), (x_5, 3)])$, which are clearly different. Thus, on this problem, the projection key is stronger than the intuitively obvious handcrafted key. The size of the keys are comparable as well.

8 Related Work

Problem specific approaches to dominance detection/subproblem equivalence are widespread in combinatorial optimization (see e.g. [7, 20]). There is also a significant body of work on caching that relies on decomposing problems into disjoint parts by fixing certain variables (e.g [11, 13]). This approach looks for equivalent projected problems but, since it does not take into account the semantics of the constraints, the approach effectively uses $D_{F \cap vars(c)}$ for every constraint c as the projection key. Thus, this approach finds strictly fewer equivalent subproblems than our approach. We could extend our approach to also split the projected problem into disjoint components but this typically does not occur in the problems of interest to us. Interestingly, [11] uses symmetry detection to detect symmetric (rather than equivalent) subproblems, but the method used does not appear to scale.

8.1 Dynamic Programming

Dynamic programming (DP) [3] 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 problem as recursive equations relating the answers to optimization problems of the same form. When applicable, this approach 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 which is defined as follows: $D \equiv \wedge_{i=1}^{n} x_i \in \{0, 1\}$, $C \equiv \sum_{i=1}^{n} w_i * x_i \leq W \wedge e > K$ where $e \equiv \sum_{i=1}^{n} p_i * x_i$ is the expression to be maximised, n is the number of available items, K is the bound on the objective (minimum profit), W is the maximum weight the knapsack can carry, w_i is the weight of item i, and p_i is the profit of item i. The DP formulation of this problem defines $knp(j, w)$ as the maximum profit achievable using the first j items with a knapsack whose maximum weight is 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.

The performance of the 0-1 knapsack problem is terrible in its natural CP formulation, it requires $O(2^n)$ search to prove the optimal solution. A CP approach with caching will represent the key as the triple (F, Rw, Rp), where F

is the set of fixed variables, Rw is an integer representing the remaining weight limit (i.e., the extra weight the knapsack can still carry), and Rp is an integer representing the remaining profit required to achieve the optimal profit U. We can see that with a fixed search ordering, F can only take on n values, the remaining weight limit can only take on W values, and the remaining profit can only take on U values. Thus, the maximum number of distinct entries in the hashtable, and hence the worse case complexity of the algorithm, is $O(nWU)$, which is much better than $O(2^n)$, but not quite as good as the DP solution.

The two approaches 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 remaining weight plus remaining profit required. The CP approach thus implements a form of DP with bounding [16], since it can detect subproblem dominance: a problem with remaining weight Rw' and remaining profit required Rp' is dominated by a problem with remaining weight $Rw \geq Rw'$ and remaining profit $Rp \leq Rp'$. The DP approach 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 remaining 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. In some sense we can think of the automatic caching approach as defining *dynamic programming for free*. Rather than determine a complex recursive equation, we simply model the problem as usual, and obtain a dynamic programming like solution.

8.2 Symmetry Breaking

Symmetry breaking aims to speed 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) [6] works by performing a "dominance check" at each search node and, if the node is found to be dominated, not exploring the node.

At first glance, SBDD may appear to be very similar to automatic caching. However, the two techniques actually exploit different kinds of dominances. SBDD detects and exploits *symmetric* subproblems, whereas automatic caching exploits *equivalent* and *dominated* subproblems. In our terminology, we could describe SBDD as follows. Let σ be a mapping from valuations of constraint problem P to valuations of P. Mapping σ is a symmetry of P if: $\sigma(\theta) \in solns(P)$ iff $\theta \in solns(P)$, i.e., if σ maps solutions to solutions. We can extend σ to a mapping from constraints to constraints as follows: $\theta \in solns(c)$ iff $\sigma(\theta) \in solns(\sigma(c))$.

Definition 6. *Let $P \equiv (C, D)$ and $P' \equiv (C, D')$ be two different subproblems arising during the search from some initial constraint problem P_0. Let $F = fixed(D)$, $U = vars(C) - F$, $F' = fixed(D')$, and $U' = vars(C) - F'$. P symmetrically dominates P' w.r.t. symmetry σ of P_0 iff*

– *$fx(D) = \sigma(fx(D'))$ (or equivalently, $D_F = \sigma(D'_{F'})$)*

– $\sigma(\exists_{F'}.(C \wedge D')) \Rightarrow \exists_F.(C \wedge D)$

It follows from the definition of entailment and symmetry that if P symmetrically dominates P' and P has no solutions, then P' has no solutions either. SBDD works by, given a current partial assignment $fx(D')$, detecting a previous failed partial assignment $fx(D)$ that is equal to $\sigma(fx(D'))$. Since $\sigma(fx(D')) = fx(D)$ implies $\sigma(\exists_{F'}.(C \wedge D')) \Rightarrow \exists_F.(C \wedge D)$, then SBDD can correctly prune P'. Automatic caching on the other hand, looks for a subproblem dominance of the form $\exists_F.(C \wedge D') \Rightarrow \exists_F.(C \wedge D)$, where $fx(D') \neq fx(D)$, which yields a different set of dominances.

Note that it is perfectly feasible, and correct, to use both automatic caching and SBDD simultaneously. At a particular node, automatic caching will prune the node if it detects that the subproblem is dominated by a previously failed subproblem, while SBDD will prune the node if it detects that it is dominated by a previously failed symmetric subproblem. We show with the following example that neither method subsumes the other.

Example 12. Consider $P_0 \equiv (C, D)$ where $C \equiv \{\mathit{alldiff}([x_1, x_2, x_3, x_4, x_5]), x_1 + 4 \leq x_4 + x_5\}$ and $D \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..5\}, x_4 \in \{1..5\}, x_5 \in \{1..5\}\}$. The problem has variable symmetry $x_4 \leftrightarrows x_5$., i.e., every valuation of P_0 is symmetric to that obtained by swapping variables x_4 and x_5.

Consider the subproblems P and P' where $fx(D) \equiv \{x_1 = 1, x_2 = 2\}$, and $fx(D') \equiv \{x_1 = 2, x_2 = 1\}$. In both cases, the *alldiff* propagates to give $x_3 \in \{3..5\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}$. The linear constraint becomes satisfied in both and has $\mathit{key}(c, D_F) \equiv \mathit{true}$. Thus, their projection keys match and automatic caching detects that P and P' are equivalent. On the other hand, there is no symmetry σ of P_0 that maps $\{x_1 = 1, x_2 = 2\}$ to $\{x_1 = 2, x_2 = 1\}$ and, thus, SBDD can do nothing here.

Consider the subproblems P and P' where $fx(D) \equiv \{x_1 = 2, x_4 = 3\}$, and $fx(D') \equiv \{x_1 = 2, x_5 = 3\}$. In P, propagation gives $x_2 = \{1, 4\}, x_3 = \{4, 5\}, x_5 = \{4, 5\}$. In P', propagation gives $x_2 = \{1, 4\}, x_3 = \{4, 5\}, x_4 = \{4, 5\}$. SBDD would detect that these two subproblems are symmetric, because the partial assignments $\{x_1 = 2, x_4 = 3\}$ and $\{x_1 = 2, x_5 = 3\}$ map to each other under $x_4 \leftrightarrows x_5$. On the other hand, automatic caching determines that they are not equivalent since $F \neq F'$ and, thus, can do nothing here.

It is also possible for both techniques to detect the same subproblem dominance. Consider the subproblems P and P' where $fx(D) \equiv \{x_4 = 3, x_5 = 4\}$, and $fx(D') \equiv \{x_4 = 4, x_5 = 3\}$. Both propagate to $x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{2, 5\}$. The linear constraint becomes satisfied in both and has $\mathit{key}(c, D_F) \equiv \mathit{true}$. Thus, their projection keys match and automatic caching detects that P and P' are equivalent. SBDD detects that they are also symmetric since $\{x_4 = 3, x_5 = 4\}$ and $\{x_4 = 4, x_5 = 3\}$ map to each other under variable symmetry $x_4 \leftrightarrows x_5$. □

It is possible to extend our caching approach to exploit problem symmetries.

Theorem 6. *Let $P \equiv (C, D)$ and $P' \equiv (C', D')$ be subproblems arising during the search from some initial constraint problem P_0. Let $F = \mathit{fixed}(D)$, $U = \mathit{vars}(C) - F$, $F' = \mathit{fixed}(D')$, and $U' = \mathit{vars}(C) - F'$. Suppose σ is a symmetry of P_0 s.t. $\forall c \in C, \sigma(c) \in C$. If $\sigma(fx(D')) = fx(D)$, $\sigma(D'_{U'}) \Rightarrow D_U$, and $\forall c \in$*

$C, \sigma(key(c, D'_{F'})) \Rightarrow key(\sigma(c), D_F)$, then P symmetrically dominates P' w.r.t. symmetry σ. □

Theorem 6 tells us that, once again, we can detect subproblem symmetric dominance by doing a piecewise comparison of constraint projection keys.

Our characterisation of subproblem symmetry in terms of projections is strictly more general than the characterisation using partial assignments. This is because $\sigma(fx(D')) = fx(D)$ implies $\sigma(\exists_{F'}.(C \wedge D')) \Rightarrow \exists_F.(C \wedge D)$, but not vice versa. Therefore, it is possible for P and P' to be symmetric subproblems even when $\sigma(fx(D')) \neq fx(D)$. For example, this can occur when conditional symmetries exist. Since SBDD only detects $\sigma(fx(D')) = fx(D)$, it only exploits a subset of the possible subproblem symmetries. Automatic caching, which directly works on subproblem projections, has the potential to exploit a strictly wider range of subproblem symmetries than SBDD. Extending automatic caching to efficiently exploit symmetries is a clear avenue of future work.

8.3 Nogood learning

Nogood learning approaches in constraint programming attempt to learn nogoods from failures and record these as new constraints in the program. Automatic caching can be considered one such technique, since we record sets of constraint projections which cause failure as nogoods. Another very successful method is Lazy Clause Generation (LCG) [15]. LCG uses clauses on atomic constraints $v = d$ and $v \leq d$ to record the reasons for failures and uses SAT techniques to efficiently manage the nogoods. LCG also exploits subproblem equivalences/dominances.

In terms of pruning power, neither technique strictly dominates the other. LCG has two advantages. Firstly, in LCG, only constraints which are directly involved in the conflict are used to produce the nogood, whereas in automatic caching, all active and non-satisfied constraints are used. This sometimes allows LCG to produce more powerful nogoods than automatic caching. Secondly, the nogoods derived by LCG can be propagated, rather than merely be checked for failure, so they can prune more. These two factors sometimes allow a much greater reduction in search space. However, this extra power comes with a high overhead. In LCG, every nogood that is derived is added as a clausal propagator. This takes $O(n)$ to propagate where n is the number of nogoods kept. In automatic caching the failure check is done in $O(1)$.

Automatic caching also has an advantage in terms of the kinds of nogoods they can express. While the language of nogoods used by LCG only allows atomic constraints of form $v = d$ and $v \leq d$ to be used as the basic components of nogoods, the constraint projections used in automatic caching can include more powerful forms. 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. LCG would express this as $x_1 \geq 1 \wedge x_2 \geq 2 \wedge x_3 \geq 3 \wedge$ other keys leads to failure, since there are no literals to represent partial sums. The nogood from automatic caching is strictly stronger. For example, $D \equiv \{x_1 = 2 \wedge x_2 = 3 \wedge x_3 = 0\}$ would fire the nogood from automatic caching, but not the one from LCG. This difference is apparent in the experimental results for 0-1 Knapsack.

Another advantage of automatic caching over LCG is in the ease of implementation. Automatic caching can be added to a CP solver by simply (1) adding to each propagator the code needed to build a representation of the propagator for inclusion in the key, as well as (2) adding a generic caching mechanism to the search. In Section 6 we showed how to do this for many constraints, and gave generic approaches to building an efficient representation of global constraints via decomposition. LCG requires much more fundamental changes to a CP solver in order to record an implication graph of the inferences made. In particular, each global constraint needs to be extended to explain its inferences, a task that is far from straightforward.

9 Experiments

We use the state of the art CP solver CHUFFED in our experiments. CHUFFED can be run as a naive CP solver (denoted as CHUFFED), as a CP solver with caching (denoted as CHUFFEDC), and as a Lazy Clause Generation solver (denoted as CHUFFEDL). We also compare against Gecode 3.2.2 [18] – widely recognized as one of the fastest constraint programming systems (to illustrate we are not optimizing a slow system). We use the `MurmurHash 2.0` hash function. We use models written in the modelling language MiniZinc [14]. This facilitates a fair comparison between the solvers, as all solvers use the same model and search strategy. Note that caching and lazy clause generation do not interfere with the search strategies used here, as all they can do is fail subtrees earlier. Thus, CHUFFED with caching or lazy clause generation always finds the same solution as the naive version CHUFFED and Gecode, and any speedup observed comes from a reduced search.

The experiments were conducted on Xeon Pro 2.4GHz processors with a 900 second timeout. Tables 1 and 2 presents the number of variables and constraints as reported by CHUFFED, the times for each solver in seconds as well as the number of (leaf) fails. For the three versions of CHUFFED, we report the maximum memory usage in Mb. For the automatic caching solver we also report the number of cache hits occurring in the search and the average key size in bytes. We discuss the results for each problem below. All the MiniZinc models and instances are available at `www.cs.mu.oz.au/~pjs/autocache/`.

9.1 Knapsack

0-1 knapsack is ideal for caching. The model has two linear constraints (one for the weight and one for the objective) and all its domains are binary. Thus, the representation of the key simply requires the set of fixed variables and two integers. The non-caching solvers 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 at all. 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, automatic caching is not as efficient as pure DP.

9.2 MOSP

The MOSP model used in the experiments is similar to that presented in Section 4, but has some additional conditional dominance breaking constraints [4] that make the (non-caching) search much faster. We use random instances from [4]. Automatic caching gives up to two orders of magnitude speedup. The speedup grows exponentially with problem size. LCG also exploits a similar amount of equivalence/dominance and achieves a similar speedup.

9.3 Blackhole

The Blackhole model used in the experiments is similar to that presented in Section 7.2, but has additional conditional symmetry breaking constraints [8]. We generated random instances and used only the hard ones for this experiment. 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 that are left. Note that the manual caching reported in [20] achieves speedups in the same range (on hard instances). LCG exploits even more dominance than automatic caching. But this extra dominance exploitation comes at such a high cost that it ends up being much slower overall.

9.4 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 subproblem symmetry is the same as that in bin packing. Suppose that a set of courses have been assigned to the first k periods and has "filled" them. The remaining subproblem only depends on the set of unassigned courses, and not on how the earlier courses were assigned to the first k periods. Any permutation of those k period assignments that satisfy the constraints lead to the same subproblem. We use the model of [10], but with some additional redundant constraints that considerably reduce the search required to find an optimal solution for caching and non-caching solvers. Note that by Theorem 4, the redundant linear constraints do not have to be considered in the projection key. 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 equivalences, giving orders of magnitude speedup. In this case, lazy clause generation exploits more equivalence/dominance and is faster.

9.5 Radiation Therapy

In the Radiation Therapy problem [2], 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 [2]. Once again both automatic caching and lazy clause generation produce orders of magnitude speedup, though lazy clause generation once again exploits more equivalence/dominance and is faster.

9.6 Memory Consumption and Overhead

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 thousands of bytes. The memory results in the table certainly show that caching requires substantially more memory than executing a standard CP solver, and somewhat more than a lazy clause generation solver. Still on a modern computer, we can usually search several hundred thousand 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. The lazy clause generation solver implements these strategies already, although they wont be applied for searchs with less than 100,000 fails.

The overhead for caching is quite variable (it can be read from the tables as the ratio of fail 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.

10 Conclusion

We have described how to automatically exploit subproblem dominances 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 dominances.

The automatic caching technique is quite robust. It can find and exploit subproblem dominances 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 many 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 dominance, and is superior on some problems, in particular those with large linear constraints. It is much easier to extend a CP system to include automatic caching, than to extend it to implement Lazy Clause Generation. In particular, we have explained how to easily generate caching schemes for arbitrary complex global constraints. Whereas adding global constraints that explain their propagations, which is required for their use in a lazy clause generation system, is considerably more difficult.

Overall automatic caching can be highly beneficial for problems involving subproblem dominance. Since it is trivial to invoke, it seems always worthwhile to try automatic caching for a particular model, and determine empirically whether it is beneficial or not.

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. This research has been funded in part by the ARC DP0879710 project.

References

1. Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.
2. D. Baatar, N. Boland, S. Brand, and P. J. Stuckey. Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In *Proc. of CPAIOR 2007*, pages 1–15, 2007.
3. R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
4. G. Chu and P.J. Stuckey. Minimizing the maximum number of open stacks by customer search. In *Proceedings of CP 2009*, pages 242–257, 2009.
5. M. Garcia de la Banda and P.J. Stuckey. Dynamic programming to minimize the maximum number of open stacks. *INFORMS JOC*, 19(4):607–617, 2007.
6. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Proceedings of CP 2001*, pages 93–107, 2001.
7. A. Fukunaga and R. Korf. Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *J. Artif. Intell. Res. (JAIR)*, 28:393–429, 2007.
8. I. Gent, T. Kelsey, S. Linton, I. McDonald, I. Miguel, and B. Smith. Conditional symmetry breaking. In *Proceedings of CP 2005*, pages 256–270, 2005.
9. I. Gent, K. Petrie, and J-F. Puget. *Handbook of Constraint Programming*, chapter Symmetry in Constraint Programming, pages 329–376. Elsevier, 2006.
10. Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Modelling a balanced academic curriculum problem. In *Proceedings of CP-AI-OR-2002*, pages 121–131, 2002.
11. M. Kitching and F. Bacchus. Symmetric component caching. In *Proceedings of IJCAI 2007*, pages 118–124, 2007.
12. Michael J. Maher. Herbrand constraint abduction. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 397–406. IEEE Computer Society, 2005.
13. R. Marinescu and R. Dechter. And/or branch-and-bound for graphical models. In *Proceedings of IJCAI 2005*, pages 224–229, 2005.

14. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Proceedings of CP 2007*, pages 529–543, 2007.
15. O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
16. J. Puchinger and P.J. Stuckey. Automating branch-and-bound for dynamic programs. In *Proceedings of PEPM 2008*, pages 81–89, 2008.
17. J.-C. Regin. Generalized arc consistency for global cardinality constraint. In *14th National Conference on Artificial Intelligence (AAAI-96)*, pages 209–215, 1996.
18. C. Schulte, M. Lagerkvist, and G. Tack. Gecode. `http://www.gecode.org/`.
19. B. Smith and I. Gent. Constraint modelling challenge report 2005. `http://www.cs.st-andrews.ac.uk/~ipg/challenge/ModelChallenge05.pdf`.
20. B.M. Smith. Caching search states in permutation problems. In *Proceedings of CP 2005,*, pages 637–651, 2005.

Table 1. Experimental Results: Knapsack, MOSP and Blackhole

Instance	Vars.	Cons.	Gecode		Chuffed			ChuffedL			ChuffedC				
			Time	Fails	Time	Fails	Mem	Time	Fails	Mem	Time	Fails	Mem	Cache hits	Key size
knapsack-20	21	2	0.01	977	0.01	1034	12	0.02	978	12	**0.01**	354	141	95	54
knapsack-30	31	2	0.38	145183	0.81	145438	12	3.42	145184	23	**0.02**	2160	142	1305	54
knapsack-40	41	2	17.9	6032500	37.80	6032644	12	168.48	6032500	33	**0.04**	3038	144	2259	59
knapsack-50	51	2	>900	–	>900	–	12	>900	–	41	**0.08**	6176	148	4810	60
knapsack-60	61	2	>900	–	>900	–	12	>900	–	44	**0.10**	7802	150	5993	60
knapsack-100	101	2	>900	–	>900	–	12	>900	–	68	**0.40**	26581	177	23042	68
knapsack-200	201	2	>900	–	>900	–	12	>900	–	125	**2.33**	110739	297	103505	80
knapsack-300	301	2	>900	–	>900	–	12	>900	–	190	**6.50**	240581	484	230470	92
knapsack-400	401	2	>900	–	>900	–	12	>900	–	255	**13.77**	393911	715	379063	104
knapsack-500	501	2	>900	–	>900	–	12	>900	–	291	**25.32**	595400	1018	576916	116
mosp-30-30-4-1	1021	1861	13.93	34427	2.98	40456	16	0.49	3061	17	**0.47**	5331	146	225	1650
mosp-30-30-2-1	1021	1861	>900	–	>900	–	16	**4.23**	21612	21	4.26	56012	148	4896	1656
mosp-40-40-10-1	1761	3281	0.93	766	0.15	814	22	0.18	655	22	**0.15**	703	153	3	2592
mosp-40-40-8-1	1761	3281	2.72	2652	0.40	2820	22	0.35	1464	22	**0.27**	1590	153	19	2737
mosp-40-40-6-1	1761	3281	40.85	46630	6.30	49904	21	1.58	6327	22	**1.26**	8632	153	288	2927
mosp-40-40-4-1	1761	3281	>900	–	328.24	2931342	20	10.56	32037	28	**7.84**	58609	156	3185	2995
mosp-40-40-2-1	1761	3281	>900	–	>900	–	20	39.28	91074	46	**36.49**	268725	173	24040	3046
mosp-50-50-10-1	2701	5101	8.62	3513	0.78	3621	30	0.73	2333	30	**0.59**	2498	162	21	3976
mosp-50-50-8-1	2701	5101	36.88	16820	3.29	18164	29	1.96	6018	30	**1.56**	7327	161	125	4118
mosp-50-50-6-1	2701	5101	>900	–	234.47	1564331	27	19.87	41436	41	**12.89**	71894	165	2815	4190
blackhole-1	104	407	101.79	425470	36.85	590628	14	91.99	111522	216	**17.90**	203018	264	24665	680
blackhole-2	104	411	59.24	273090	21.19	313178	14	72.28	99595	182	**14.12**	161808	229	11199	650
blackhole-3	104	434	31.00	132406	25.02	541716	14	58.26	94384	176	**17.63**	299079	283	16704	598
blackhole-4	104	393	68.32	324063	27.77	500890	14	87.13	118694	193	**15.42**	196622	249	12886	631
blackhole-5	104	429	411.83	2032279	155.28	2654106	14	146.55	162946	201	**66.41**	840335	276	95166	672
blackhole-6	104	448	157.30	739404	57.60	882158	14	43.02	75068	150	**24.14**	255541	206	22362	655
blackhole-7	104	407	84.68	482083	32.43	642540	14	177.17	194795	206	**11.05**	125747	322	12078	632
blackhole-8	104	380	125.83	628861	47.27	777901	14	112.12	149123	192	**27.35**	313000	300	40262	646
blackhole-9	104	404	87.93	385227	42.69	724076	14	219.70	264754	205	**23.55**	295363	388	25839	658
blackhole-10	104	364	211.86	1035902	91.85	1567065	14	391.60	395195	218	**38.09**	445726	605	48605	640

Table 2. Experimental Results: BACP and Radiation

Instance	Vars.	Cons.	Gecode		Chuffed			ChuffedL			ChuffedC				
			Time	Fails	Time	Fails	Mem	Time	Fails	Mem	Time	Fails	Mem	Cache hits	Key size
curriculum_8	838	1942	0.01	0	0.01	0	13	**0.01**	0	13	0.01	0	142	0	626
curriculum_10	942	2214	0.01	1	0.01	1	14	**0.01**	1	14	0.01	1	142	0	714
curriculum_12	1733	4121	0.01	0	0.01	0	15	**0.01**	1	16	0.01	0	144	0	1012
bacp-medium-1	1121	2654	29.47	404779	35.21	404779	14	**5.95**	26764	19	11.59	133456	322	18505	1045
bacp-medium-2	1122	2650	>900	–	>900	–	14	**0.01**	41	15	9.53	120186	298	71575	902
bacp-medium-3	1121	2648	462.64	8330028	383.81	8330028	14	**0.03**	105	15	2.40	43886	196	40973	951
bacp-medium-4	1119	2644	5.76	101650	4.59	101650	14	0.69	5441	14	**0.61**	10084	153	3178	869
bacp-medium-5	1119	2641	54.17	896859	56.90	896859	14	**0.30**	1783	14	2.41	33884	179	12688	867
bacp-hard-1	1121	2655	>900	–	>900	–	14	**0.01**	5	14	55.94	727847	1112	479361	924
bacp-hard-2	1118	2651	>900	–	>900	–	14	**0.01**	8	14	191.78	1518859	2389	669450	1039
radiation-6-9-1	877	942	>900	–	>900	–	13	**1.39**	4290	22	11.37	184133	809	122799	2597
radiation-6-9-2	877	942	>900	–	>900	–	13	**2.68**	7835	22	25.03	431953	1478	285552	2192
radiation-7-8-1	1076	1168	>900	–	>900	–	14	0.31	1361	29	**0.27**	2748	158	393	3782
radiation-7-8-2	1076	1168	188.12	3940867	177.17	7881662	13	**0.22**	808	29	0.38	4467	163	2647	3030
radiation-7-9-1	1210	1301	312.09	5518905	283.70	11032846	13	**0.78**	3341	32	1.33	14993	226	7426	3814
radiation-7-9-2	1210	1301	140.44	2316137	110.82	4631769	14	**3.13**	18207	34	4.43	53683	446	14491	4068
radiation-8-9-1	1597	1718	>900	–	>900	–	14	**1.89**	5232	42	23.92	227154	1831	152720	5686
radiation-8-9-2	1597	1718	>900	–	>900	–	14	**2.04**	6518	43	9.71	107545	702	56586	3908
radiation-8-10-1	1774	1894	9.73	101062	8.95	201703	14	**8.63**	36612	48	10.54	88663	995	10980	7267
radiation-8-10-2	1774	1894	>900	–	>900	–	15	**5.75**	10322	44	45.52	385431	2114	233590	3987