

# Enhancing Constraint Models of Planning Problems by Common Subexpression Elimination

**Andrea Rendl, Ian Miguel, Ian P. Gent**

School of Computer Science, University of St Andrews, UK  
{andrea, ipg, ianm}@cs.st-andrews.ac.uk

**Peter Gregory**

University of Strathclyde, Glasgow, UK  
pg@cis.strath.ac.uk

## Abstract

Constraint Programming is an attractive approach for solving AI planning problems by modelling them as Constraint Satisfaction Problems (CSPs). However, formulating effective constraint models of complex planning problems is challenging, and CSPs resulting from standard approaches often require further enhancement to perform well. Common subexpression elimination is a general technique for improving constraint models, which has been shown to lead to a great reduction in instance size, solving time and search space. This paper argues that models of AI planning problems are particularly amenable to this approach. We present four case studies to substantiate this argument, three of which include novel constraint models of AI planning problems. In each, we describe the constraint model, highlight the sources of common subexpressions, and present an empirical analysis of the effect of eliminating common subexpressions.

## Introduction

AI planning is an active, long-established research area, with a wide applicability to such diverse tasks as automating data-processing procedures, game-playing, and large-scale logistics problems. The classical AI Planning problem is to find a sequence of actions (a plan) to transform an initial world state into a goal world state. We consider solving AI Planning problems using constraint solving, a powerful technique for tackling hard combinatorial problems. Constraint solving proceeds in two phases. First, the problem is *modelled* as a set of decision variables, and a set of constraints on those variables that a solution must satisfy. Second, a constraint solver is used to search for solutions to the model: assignments of values to variables satisfying all constraints.

Common subexpression elimination is an efficient general technique for transforming a constraint model into one which requires less effort to solve. It has been shown to lead to a great reduction in instance size, solving time, and search space (Gent, Miguel, & Rendl 2008). This paper argues that models of AI planning problems are particularly amenable to this approach. We present four case studies to substantiate this argument, three of which include novel constraint models of AI planning problems. In each, we describe the constraint model, highlight the sources of common subexpressions, and present an empirical analysis of the effect of eliminating common subexpressions.

## Background

Constraint modelling and solving of planning problems has been studied in the context of many systems, such as CPlan (van Beek & Chen 1999), the planning & scheduling framework in (Garrido 2006), the temporal POCL planner CPT (Vidal & Geffner 2006) or the distributed multi-agent system in (Sapena *et al.* 2008). Hence, there exist many different approaches on how to model a planning problem as a CSP. (Barták & Toropila 2008) describe three different constraint models for planning which are derived from different successful modelling approaches. They all share the same basic set of constraint variables:

- $v(n - 1)$  **state variables**  $V_i^s$ , representing the state of the world at step  $s$ , where  $v$  is the number of properties of a state,  $n$  the length of the plan,  $i$  ranges over the state properties (i.e. from 0 to  $v$ ) and  $s$  ranges from 0 to  $n - 1$ .
- $n$  **action variables**  $A^s$ , representing the action chosen at step  $s$ , where  $s$  ranges from 0 to  $n - 1$

State and action variables are connected by logical constraints that summarise the chosen action's *preconditions* and *effects* on the state variables, as well as *frame axioms* (i.e. constraints that enforce that certain state properties stay the same during a state transition). Each of the three models differ in how preconditions, effects and frame axioms are represented. The basic model describes preconditions and effects by the two constraints

$$(A^s = act) \Rightarrow \text{Pre}(act)^s \quad \forall act \in \text{Dom}(A^s) \quad (1)$$

$$(A^s = act) \Rightarrow \text{Eff}(act)^{s+1} \quad \forall act \in \text{Dom}(A^s) \quad (2)$$

stating that if action  $act$  is chosen at step  $s$ , then precondition  $\text{Pre}(act)^s$  and effect  $\text{Eff}(act)^{s+1}$  have to hold.  $\text{Pre}(act)^s$  and  $\text{Eff}(act)^{s+1}$  are a conjunction of conditions where appropriate state variables are set to  $act$ 's preconditions at step  $s$ , and effects in step  $s+1$ , respectively. The frame axiom

$$A^s \in \text{NonAffAct}(V_i) \Rightarrow (V_i^s = V_i^{s+1}), \quad \forall i \in (0, v-1) \quad (3)$$

states that if action  $A^s$  has no effect on state property  $i$  then  $V_i^s$  and  $V_i^{s+1}$  are equal.

The second model represents supporting actions (Do & Kambhampati 2000) by adding variables  $S_i^s$  that indicate the action responsible for the value of state property  $i$ . Preconditions are formulated as in (1) but effects(4) and frame axioms(5) are stated as

$$(S_i^s = act) \Rightarrow (V_i^s = val) \quad \forall act \in \text{Dom}(S_i^s) \quad (4)$$

$$(S_i^s = \text{no-op}) \Rightarrow (V_i^s = V_i^{s+1}) \quad (5)$$

In the third model, effects and frame axioms are merged into so-called successor-state constraints (Lopez & Bacchus 2003). Successor state constraints state that a state variable has value  $val$  at step  $s$  only if either an action has changed it or it was the same in the previous step, formally

$$V_i^s = val \leftrightarrow (A^{s-1} \in C(i, val)) \vee (V_i^{s-1} = val \wedge A^{s-1} \in N(i)) \quad (6)$$

where  $C(i, val)$  is the set of actions that effect  $V_i^s = val$  and  $N(i)$  is the set of actions that do not affect  $V_i$ .

Throughout, we consider the number of steps of the plan to be a parameter to the constraint model. Hence, to find an optimal solution to a given planning problem, one would iteratively increase the value of the steps parameter of the corresponding constraint model until a solution is found.

### Common Subexpressions

Constraint models can be enhanced automatically by eliminating *common subexpressions* (Gent, Miguel, & Rendl 2008). Two (sub)expressions are *common* if they are the same under all satisfying variable assignments. For instance, the two constraints  $A \Rightarrow (x=0)$  and  $B \Rightarrow (x=0)$  contain the common subexpression  $x=0$ . Common subexpressions are eliminated during *flattening*, i.e. the decomposition of complex expressions into simpler expressions (to adapt them to the target solver). Consider the constraint  $x=0 \wedge y=0$  that is typically flattened to the constraints  $x=0 \leftrightarrow aux_1$ ,  $y=0 \leftrightarrow aux_2$ ,  $aux_1 \wedge aux_2$ , where the newly introduced variables  $aux_1$  and  $aux_2$  are called *auxiliary variables*. Common subexpressions are eliminated by representing each occurrence of an expression with the same auxiliary variable, instead of introducing several variables for the same expression and is a computationally cheap process to perform.

Common subexpressions in constraint models of planning problems originate in constraints corresponding to effects, preconditions and frame axioms. Our hypothesis is that a model contains common subexpressions if two or more actions share conditions in preconditions, effects or frame axioms. As an example, consider a planning problem with two actions  $act_1$  and  $act_2$  with preconditions  $\text{Pre}(act_1)^s = a \wedge b \wedge c$  and  $\text{Pre}(act_2)^s = b \wedge d$  where  $a, b, c, d$  are conditions that restrict the state variables, e.g.  $a : V_i^s = 0$ . As both preconditions share the condition  $b$ , the basic precondition constraint in Equation (1) will share arguments, hence contain common subexpression  $b$ :

$$(A^s = act_1) \Rightarrow (a \wedge b \wedge c)^s$$

$$(A^s = act_2) \Rightarrow (b \wedge d)^s$$

The same holds if  $act_1$  and  $act_2$  share a condition in effects  $\text{Eff}(act_1)$  and  $\text{Eff}(act_2)$ , when representing effects by Equation (2). Furthermore, in the effect formulation using supporting actions in Equation (4) we get the constraints

$$(S_i^s = act_1) \Rightarrow (V_i^s = val)$$

$$(S_i^s = act_2) \Rightarrow (V_i^s = val)$$

where the common subexpression  $V_i^s = val$  is the effect shared by  $act_1$  and  $act_2$ . Similarly, actions sharing frame

axioms create common subexpressions. Assume both actions  $act_1$  and  $act_2$  leave the state property  $V_i$  unchanged. Then the corresponding frame axioms (see Equation 3) contains the common subexpression  $V_i^s = V_i^{s+1}$ :

$$(A^s = act_1) \Rightarrow (V_i^s = V_i^{s+1}), \quad (A^s = act_2) \Rightarrow (V_i^s = V_i^{s+1})$$

When effects and frame axioms are merged into Equation (6), we detect common subexpressions  $V_i^s = val$  when posting the constraint for consecutive steps  $s$  and  $s-1$ :

$$V_i^s = val \leftrightarrow (A^s \in C(i, val)) \vee (V_i^{s-1} = val \wedge A^{s-1} \in N(i))$$

$$V_i^{s+1} = val \leftrightarrow (A^{s+1} \in C(i, val)) \vee (V_i^s = val \wedge A^s \in N(i))$$

Eliminating a common subexpression, i.e. representing each occurrence by the same auxiliary variable during flattening, saves one variable and one constraint per occurrence, which reduces the resulting constraint instance. In the following sections, we discuss four planning problems where, when modelled as CSPs, common subexpressions arise. In the early examples, the degree of overlap among preconditions, effects and frame conditions is small, and there are correspondingly few common subexpressions. In the later examples the opposite is true. We investigate the sources of common subexpressions and the impact of their elimination.

### Case Study: Sokoban

The well known Sokoban puzzle/game is played in a 2D virtual warehouse. The problem is to find a sequence of horizontal and vertical moves for the sokoban such that every crate is in a goal cell. The sokoban can push a single crate one cell in the direction that he moves. Neither the sokoban nor the crates can move onto a wall cell, and no two objects (sokoban or crate) can occupy the same cell.

Our constraint model is shown in Fig.1. The parameters are the width  $w$  of the grid, the total number  $n$  of cells, the initial positions of the sokoban ( $p_{init}$ ) and *crates*, the *goal* positions for the crates, and the positions of the *walls*.

State is given simply by the positions of the sokoban and crates. We use a single variable for each, per time step (*sok-Posn*, *cratePosns*). The domain of each represents the cells of the warehouse, enumerated row-wise left to right. Another possibility would be to have a pair of variables representing the coordinate position of the sokoban/crate, but our formulation allows us to model moving the sokoban and pushing the crates very simply, as described below. This state is initialised by constraints (1) and (2). Constraints (3) and (4) prevent either the sokoban or the crates from ever entering a wall cell. Constraint (5) prevents any two crates from being co-located (that the sokoban can never occupy the same cell as a crate is implied by the push constraints below). The goal is captured by constraint (6).

A single action is possible at each step: the move made by the sokoban. We model this using an array of variables, *move*, indexed by the time step. The domain of each *move* variable contains four elements, representing the four directions in which the sokoban can move. Given the row-wise enumeration of the cells, a move left (right) decreases (increases) the cell index by 1, while a move up (down) decreases (increases) the cell index by  $w$ . Hence, movement can be modelled as a simple summation (7). Furthermore,

<pre> <b>given</b> w, n, pInit, stps:int <b>given</b> walls:matrix indexed by [WINDICES] of int(0..n-1) <b>given</b> crates:matrix indexed by [CINDICES] of int(0..n-1) <b>given</b> goals:matrix indexed by [CINDICES] of int(0..n-1) </pre>
<pre> <b>letting</b> MOVES be domain int(-w,-1,1,w) <b>letting</b> STEPS be domain int(0..stps-1) </pre>
<pre> <b>find</b> sokPosn:matrix indexed by [STEPS] of int(0..n-1) <b>find</b> cratePosns:matrix indexed by [STEPS,CINDICES] <b>find</b> move:matrix indexed by [int(0..stps-2)] of MOVES   of int(0..n-1) <b>find</b> crateMoved:matrix indexed by [STEPS] of bool </pre>
<pre> <b>such that</b> 1 sokPosn[0] = pInit, 2 <b>forall</b> c:CINDICES.cratePosns[0,c] = crates[c], 3 <b>forall</b> s:STEPS. <b>forall</b> wall:WINDICES.   sokPosn[s] != walls[wall], 4 <b>forall</b> s:STEPS. <b>forall</b> c:CINDICES. <b>forall</b> wall:WINDICES.   cratePosns[s,c] != walls[wall], 5 <b>forall</b> s:STEPS. alldifferent(cratePosns[s,..]), 6 <b>forall</b> c:CINDICES. <b>exists</b> g:CINDICES.   cratePosns[stps-1,c] = goals[g] 7 <b>forall</b> s:int(0..stps-2).   sokPosn[s+1] = (sokPosn[s] + move[s]), 8 <b>forall</b> s:int(0..stps-2). <b>forall</b> c:CINDICES.   (sokPosn[s+1] = cratePosns[s,c]) =&gt;   (cratePosns[s+1,c] = cratePosns[s,c] + move[s]), 9 <b>forall</b> s:int(0..stps-2). <b>forall</b> c:CINDICES.   ((sokPosn[s+1] = cratePosns[s,c]) ∨   (cratePosns[s+1,c] = cratePosns[s,c])), 10 crateMoved[0] = 0, 11 <b>forall</b> s:int(1..stps-1).crateMoved[s] =   (<b>exists</b> c:CINDICES.sokPosn[s] = cratePosns[s-1,c]), 12 <b>forall</b> s1:int(0..stps-2). <b>forall</b> s2:int(s1+1..stps-1).   (sokPosn[s1] = sokPosn[s2]) =&gt;   ((<b>sum</b> s3:int(s1..s2).crateMoved[s3]) &gt; 0), </pre>

Figure 1: Constraint model of Sokoban in ESSENCE’.

pushing crates can be modelled in the same way, simply by checking whether the sokoban at step  $s + 1$  is in a cell occupied by a crate at step  $s$  (8). Frame constraints (9), ensure that crates not pushed stay in the same cell. Finally, we exploit that, if no crate is pushed, it is pointless for the sokoban to re-visit the same cell. We introduce a Boolean variable per time step (*crateMoved*), which is true if some crate is pushed (10,11). Then, we allow the sokoban to revisit the same cell only if some crate moved in the interim (12).

The simple effect constraints and frame axioms lead to a small number of common subexpressions in Sokoban. The only source of common subexpressions is the case when a sokoban pushes a crate  $c$  at step  $s$ , stated by  $\text{sokPosn}[s+1] = \text{cratePosns}[s,c]$  This condition occurs in constraint (8) that describes the effect of pushing a crate, in the frame constraint (9), and in the implied constraint (11) that restricts the sokoban to go in circles only when moving a crate. In summary, we have  $(s - 1) * c$  common subexpressions where  $s$  is the length of the plan and  $c$  the number of crates.

## Case Study: Settlers

The Settlers problem, introduced in the third International Planning Competition (Long & Fox 2003), is loosely based on the German board game ‘Die Siedler von Catan’. Each

<pre> 1 <b>forall</b> city:CITIES . <b>forall</b> s:STEPS .   (production[s, city,ORE] &gt; 0) =&gt;   (buildingBuiltInCity[city,MINE] &lt; s) </pre>
<pre> 2 <b>forall</b> city:CITIES . <b>forall</b> s:STEPS .   (totalExport[s,city,ORE] &gt; 0) =&gt;   (buildingBuiltInCity[city,MINE] &lt; (s-1)) </pre>
<pre> 3 <b>forall</b> city:CITIES . <b>forall</b> good:GOODS . <b>forall</b> s:STEPS .   buildingRequirement[s, city, good] =   <b>sum</b> building:BUILDINGS .   (buildingBuiltInCity[city,building] = s) *   requirementTable[building,good]   + houses[city] * houseRequirements[good] * (s=horizon) </pre>
<pre> 4 <b>forall</b> s:STEPS . <b>forall</b> city:CITIES .   totalLabour[s,city] =   <b>sum</b> building:BUILDINGS .   (buildingBuiltInCity[city,building] = s) *   labour[building]   + (<b>sum</b> good:GOODS .   production[s,city,good]*labour[good]) </pre>

Figure 2: Selection constraints of the Settlers Model in ESSENCE’.

instance has a goal of constructing various buildings across a set of cities. Different cities have access to different raw-materials hence some goods have to be transported between cities in order to construct the required buildings. There are three ways of transporting goods: by cart, train and ship. There are different costs(labour) associated with creating and operating these forms of transport.

Fig. 2 shows a selection of constraints of the constraint model of Settlers (Gregory & Rendl 2008). The maximum number of steps, the amount of cities and building requirements are given as parameters. Array *buildingBuiltInCity*[ $c,b$ ] represents the time step in which building  $b$  was built in city  $c$ . The array *buildingRequirement*[ $s,c,g$ ] contains the units of good  $g$  (e.g. iron) required to construct a building in city  $c$  at time step  $s$ . For every city and step, we constrain *buildingRequirement* to equal the sum of goods needed to build construction sites and houses (constraint (3)).

The production of material  $m$  in city  $c$  at step  $s$  is represented by the variable-array *production*[ $s,c,m$ ]. *import* and *export* of material are represented in the same manner. To produce a particular material, the appropriate production site must have been built before. For instance, to produce ore at time  $s$ , a mine must have been built before  $s$ , (see constraint (1)). The total labour in each time step  $s$  in city  $c$  is given by the variable-array *totalLabour*[ $s,c$ ] and restricted by constraint (4). We measure the quality of a plan for Settlers by the amount of labour required.

There are two main sources of common subexpressions. The first source results from a shared precondition of production and export of a particular good at step  $s$ : the appropriate production facility must have been built before  $s$ . Fig. 2 shows the production constraint(1) and export constraint(2) concerning ore production. Note that both constraints share subexpressions of the form ‘*buildingBuiltInCity*[ $city,MINE$ ] <  $s$ ’ stating the precondition that a mine in *city* was built before step  $s$ . This kind of common subexpression occurs for every city, step and production facility (e.g. mine), thus  $c * s * 5$  times where

$c$  is the number of cities,  $s$  the number of steps and 5 the number of production facilities.

The second source of common subexpressions arises because the construction of a building  $b$  in city  $c$  increases both the requirement of goods in  $c$  and the amount of labour for building  $b$ . Constraint (3) in Fig.2 describes the good requirements, constraint (4) the amount of labour in each city. Both constraints share the subexpression ‘buildingBuiltInCity[ $c, b$ ] =  $s$ ’ stating that  $b$  was built in  $c$  at step  $s$ . In each problem instance, there are  $5 * c * s$  common subexpressions of this form, where  $c$  is the number of cities,  $s$  the number of steps and 5 the number of production facilities.

### Case Study: Peg Solitaire

Peg Solitaire (see CSPLib 37) is played on a board with a number of holes. In the English version of the game, the board is in the shape of a cross with 33 holes. Pegs are arranged on the board so that at least one hole remains. Moves are draughts/checkers-like and are horizontal or vertical. There are several variations of peg solitaire. We focus on the classic *reversal* game in which an initial state with just one peg missing is transformed into a state with a single peg remaining in the same location as the initial hole.

Our constraint models are from (Jefferson *et al.* 2006), with two sets of variables. First, the 1-dimensional array *moves* represents the action chosen at every time step. In our variant, it is indexed by a fixed length of 31 time steps and the variables’ domain ranges over all 76 possible moves. Second, the 2-dimensional array *board* represents the state of each cell on the board at every time step. It is indexed by the board cells and time steps, and consists of 0/1 variables representing a peg (1) or an empty cell (0).

Based upon these variables, we consider two model variants (Fig. 3). The first is *action-centric*: for a given move it describes the cells that change and those that stay the same (1). The second is *state-centric*: for each cell, it describes the moves that cause it to change state and those that leave it unaffected (2). Both models share initial/goal constraints and the implied constraint. The action-centric and state-centric model correspond to the basic model and model using successor state constraints, respectively, discussed above.

The representation of a state in Peg Solitaire is more complex than for Sokoban, consisting as it does of 33 Boolean variables per step. Since each move affects three cells on the board (and leaves 30 unchanged), there is considerable overlap among the set of moves. It is from this overlap that the common subexpressions in the frame/effect/precondition constraints stem. Consider, for example, the subexpression  $board[s, c1] > board[s+1, c1]$ , which models the situation where a peg is removed from cell  $c1$  during the transition from step  $s$  to  $s + 1$ . The removal of a peg from cell  $c1$  can result from different moves, i.e. it is a shared effect. Likewise, the reverse action, placing a peg into a hole, is shared among different moves. However, the biggest overlap occurs in the frame axioms, as a particular cell is left unchanged by many different actions.

In summary, for each cell, the removal of a peg is shared by up to 8 different moves, the insertion of a peg by up to

	<b>given</b> reversal:int
	<b>letting</b> CELLS be domain int(1..33)
	<b>letting</b> moveNumber, start, middle, end ... \$ Lookup tables
	<b>find</b> moves:matrix indexed by [int(0..30)] of int(1..76)
	<b>find</b> board:matrix indexed by [int(0..31), CELLS] of bool
	<b>such that</b> \$ Initial & goal states
	<b>forall</b> c: CELLS. (c != reversal) $\Rightarrow$ (board[0,c] = true), board[0,reversal] = false,
	<b>forall</b> c: CELLS. (c != reversal) $\Rightarrow$ (board[31,c] = false), board[31,reversal] = true,
	\$ Implied: the number of pegs decreases every step
	<b>forall</b> s : STEPS . 32-s = (sum i : FIELDS . bState[s,i])
1	\$ Action-centric constraints
	<b>forall</b> s:int(0..30). <b>forall</b> c1,c2:CELLS. (moves[s] = moveNumber[c1,c2]) $\Rightarrow$ (board[s,c1] > board[s+1,c1]) $\wedge$ (board[s,middle[c1,c2]] > board[s+1,middle[c1,c2]]) $\wedge$ (board[s,c2] < board[s+1,c2]))
2	\$ Alternative state-centric constraints
	<b>forall</b> s:int(0..30). <b>forall</b> c:CELLS. (board[s,c] > board[s+1,c]) $\Leftrightarrow$ \$ peg removal ( <b>exists</b> c1,c2:CELLS. (c1 != c2) $\wedge$ (moves[s] = moveNumber[c1,c2]) $\wedge$ ((c = start[c1,c2]) $\vee$ (c = middle[c1,c2])))
	<b>forall</b> s:int(0..30). <b>forall</b> c:CELLS. (board[s,c] < board[s+1,c]) $\Leftrightarrow$ \$ peg insertion ( <b>exists</b> c1,c2:CELLS. (c1 != c2) $\wedge$ (moves[s] = moveNumber[c1,c2]) $\wedge$ (c = end[c1,c2]))

Figure 3: Action-centric (1) and State-centric (2) constraint models of peg solitaire reversals in ESSENCE’.

4 different moves, and up to 72 different moves leave the cell unchanged. These three overlaps are the main sources of common subexpressions in our Peg Solitaire model. Specifically, a standard instance of the action-centric model has 3,999 common subexpressions, which when eliminated, save 75,857 auxiliary variables. A typical instance of the state-centric model contains 5,890 common subexpressions, which when eliminated, save 30,039 auxiliary variables. The difference between the number of common subexpressions of both models lies in the different representations of frame axioms and preconditions and effects of actions.

### Case Study: Plotting

Plotting is a puzzle game made by Taito in 1989, see Fig. 4. It is played on a 14x14 grid, where the perimeter is composed of solid wall cells. The sub-grid on the bottom-right of the play area contains an arrangement of blocks of one of four types (for simplicity we exclude a fifth, wildcard, type from the grid). The player avatar can move up and down the first column. The avatar carries a single block of one of the types. It can throw this block horizontally along the row it occupies. At the start of the game, the avatar carries a wildcard. The effects of throwing a block against a wall are:

- If it hits a wall as it is travelling right, it falls vertically downwards. Additional walls are arranged to facilitate



Figure 4: Screenshot of Taito’s Plotting game.

hitting the blocks from above, as shown in the figure. This arrangement varies with instances of the puzzle — in harder instances wall cells are placed so as to prevent throwing blocks along some rows and columns.

- If it falls onto a wall, it rebounds into the avatar’s hand.

A thrown wildcard transforms into the same type as the first block it hits. For the other block types:

- If the first block a thrown block hits is of a different type from itself it rebounds into the avatar’s hand.
- If a block A hits a block B of the same type, B is consumed and A continues to travel in the same direction. All blocks above B fall one grid cell each.
- If a thrown block A, having already consumed a block of the same type, hits a block B of a different type, A replaces B, and B rebounds into the avatar’s hand.

If, after making a throw, the block that rebounds into the avatar’s hand is such that there is now no possible throw that can further reduce the blocks, the player loses a life and the block in the avatar’s hand is transformed into a wildcard block. The game is over if the player has no lives remaining. The aim of the game is to reduce the initial configuration of blocks so that at most some specified number remain.

Plotting can be seen as a planning problem. Our model captures an attempt to find a mistake-free solution to an instance, so moves leading to a loss of life will not be allowed. As well as the number of steps in the plan, the remaining parameters are: the width  $k$  and height  $r$  of the grid of blocks; the initial contents of the grid; the number of steps allowed  $s$ ; the goal number of blocks remaining; and the number of block types (a generalisation of the original problem).

A single action is possible at each step. We abstract away the use of the wall cells, and assume simply that the avatar throws a block either along one of the  $r$  rows or down one of the  $k$  columns. This is modelled using a pair of variables per time step,  $throw$  with domain  $\{0, \dots, r\}$  and  $tcoll$  with domain  $\{0, \dots, k\}$ . The 0 value is used to record that no block was thrown along a row (or column) at this time step. A simple constraint ensures that exactly one of this pair of variables takes the value one at each time step.

The avatar’s hand is represented by a single variable  $hand$  per time step the domain of which is the available block types. We model the initial wildcard simply by leaving  $hand[0]$  unconstrained. After the first move, constraint propagation ensures that it is transformed appropriately. The

grid state is represented in a position-centric manner: a two-dimensional array of variables  $grid$  per time step with domain corresponding to the available block types (represented by integers and including a 0 value to denote ‘empty’).

The goal state is represented by constraining  $grid[s]$  to contain a sufficient number of 0s that the number of blocks remaining condition is met. We constrain each move to be useful (remove at least one block) by insisting that the sum of each  $grid[i]$  is less than that of  $grid[i - 1]$ .

The remaining constraints fall into four categories: those that describe how grid cells remain unchanged, those that describe how the hand remains unchanged, those that describe how a grid cell becomes empty, and those that describe how a grid cell changes type (including a block being exchanged with that from the hand).

Plotting is the most complex of our case studies and has the most common subexpressions. We summarise the most important sources of common subexpressions:

- *Cell status*: many common subexpressions stem from the shared condition that a cell is empty at step  $s$ . It is an effect of hitting blocks, a precondition for hitting consecutive blocks, and also contained in the frame axiom that an empty cell will always stay empty. Similarly, the opposite condition, that a cell contains a block at step  $s$  is shared among effects and preconditions.
- *Throwing blocks*: the precondition that a block is thrown from a (particular) row or column is shared among several actions, such as aiming for a particular wall or block type.
- *Frame axioms*: many cells are unaffected by different shots, another source of common subexpressions.
- *Comparing block types*: the precondition that the block in the avatar’s hand is the same as a particular block in the grid applies to different actions on the grid. The opposite precondition, that the types differ, is also shared by different actions.
- *Shared conjunctions of conditions*: there are several conjunctions of the above mentioned conditions that form another big group of common subexpressions. For instance, the conjunction of “*cell(1,4)* is not empty” and “*cell(4,1)* has the same block type as the hand” is shared among the action “shoot from row 4 at *cell(4,1)*” and the action “shoot from column 1 at *cell(4,1)*”.

## Experimental Results

We formulated each model in the solver-independent modelling language ESSENCE’ and used Tailor v0.2 (Gent, Miguel & Rendl 2007) to flatten each instance for input to the constraint solver Minion v0.7.0 (Gent, Jefferson & Miguel 2006). Tailor provides optional common subexpression elimination, hence for every problem instance, we generate one file with common subexpression elimination and one without. We solve both instances with the same branching and search heuristics on the same machine (dual-Xeon 5430, 2.66GHz, 8GB RAM, Linux 2.6.18-92.1.13.el5).

Results are shown in Fig. 5. Elimination is highly effective on many instances. We see significant run time improvements in the Plotting problem and both models of Peg Solitaire. Each of these families can give an  $8\times$  or better

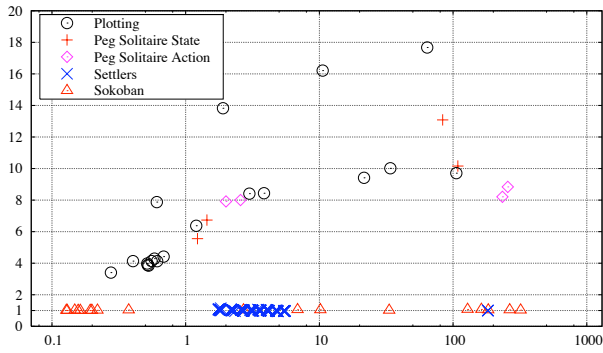


Figure 5: Speedups obtained by subexpression elimination. The (logarithmic)  $x$ -axis represents the solving time *with* elimination. The  $y$ -axis gives the factor to multiply this by to obtain the solving time *without* elimination. Points above  $y = 1$  represent instances which solves faster with elimination than without. Flattening time is excluded but is usually similar with or without elimination.

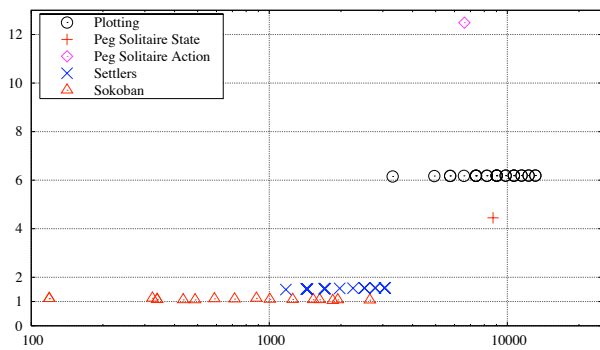


Figure 6: The  $x$ -axis represents the number of auxiliary variables introduced *with* common subexpression elimination, and the  $y$ -axis the factor reduction over not using elimination. All points are above  $y = 1$ , so we always use less variables when using elimination. All Peg Solitaire instances of a given type have the same number of auxiliary variables.

speedup. The speedups generally improve with problem difficulty. Benefits are slight on the Sokoban instances, ranging from no improvement to only a 7% speedup, although the speedup did improve slightly as problems got harder. For Settlers we saw mixed results: while we did get up to a 15% speedup, a few instances ran up to 6% slower when elimination was used. This slight slowdown may be due to fluctuations in performance from run to run, or detailed features of Minion performing differently on the different instances. Speedups are mostly due to reduction in work for the same nodecount. Most instances took the same number of search nodes with and without elimination. The exceptions were the state model of Peg Solitaire, and Plotting. For those Solitaire instances, nodes searched was reduced by about 2.5 times, so even here we see the runtime reduction was greater than the nodecount. A small number of Plotting instances showed a tiny reduction in search. This reduction in nodes searched due to common subexpression elimination has been noted and explained before (Gent, Miguel, & Rendl 2008).

We compared the size of problem instances with and without common subexpression elimination. We first looked at the number of auxiliary variables. Results in Fig. 6 show that we always use fewer extra variables this way. The smallest factor is 1.03, i.e. a 3% improvement, and the largest represents a factor of  $12.5\times$  fewer auxiliary variables. It is particularly interesting that the reduction in each family is very consistent across problem size. We obtained similar results (not illustrated) when we looked at the number of constraints in each instance. Again results were consistent in each family, the maximum factor being  $9.4\times$ . We observe that there is, as we expected, a strong correlation between families for which we obtain large reductions in the size of problems, and for which we obtain good runtime improvements.

## Discussion & Conclusions

We have considered how constraint models of planning problems can be enhanced via elimination of common subexpressions. We examined four case studies, in which constraint models were formulated using standard techniques. We found strong supporting evidence for a correlation between the degree of overlap among precondition, effect, and frame constraints and the number of common subexpressions in the constraint model. For the problems that exhibited the greatest overlap, Peg Solitaire and Plotting, the difference the reduction in solving time when common subexpressions were eliminated was dramatic and could be an order of magnitude speedup.

(Barták & Toropila 2008) suggest the use of table constraints to express preconditions/effects/frame axioms. While this would eliminate many common subexpressions, it is not always feasible. Consider the Plotting problem. The number of state variables involved in, for example, action preconditions would mean that the table constraints required would have a very high arity and would therefore be very cumbersome to specify and to propagate.

## References

- A. Garrido, E. Onaindia, M. Arangu. 2006. Using constraint programming to model complex plans in an integrated approach for planning and scheduling. *PLANSIG*, 137-144.
- Barták, R., and Toropila, D. 2008. Reformulating constraint models for classical planning. Wilson, D., and Lane, H. C., eds., *FLAIRS*, 525-530.
- van Beek, P., and Chen, X. 1999. Cplan: a constraint programming approach to planning. *AAAI '99*, 585-590.
- Do, M. B., and Kambhampati, S. 2000. Solving planning-graph by compiling it into csp. *AIPS*, 82-91.
- I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. *ECAI*, 98-102, 2006.
- I.P. Gent, I. Miguel and A. Rendl. Tailoring Solver-independent Constraint Models: A Case Study with Essence' and Minion In *SARA*, 184-199, 2007.
- Gent, I. P.; Miguel, I.; and Rendl, A. 2008. Common subexpression elimination in automated constraint modelling. *Workshop on Modeling and Solving Problems with Constraints*, 24-30.
- Gregory, P., and Rendl, A. 2008. A constraint model for the settlers planning domain. *PLANSIG*, 41-49.
- Jefferson, C.; Miguel, A.; Miguel, I.; and Tarim, A. 2006. Modelling and solving english peg solitaire. *Computers and Operations Research* 33(10):2935-2959.
- Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of AI Research* 20:1-59.
- Lopez, A., and Bacchus, F. 2003. Generalizing graphplan by formulating planning as a csp. *IJCAI*, 954-960.
- Sapena, O.; Onaindia, E.; Garrido, A.; Arangu, M. 2008. A distributed csp approach for collaborative planning systems. *Eng. Appl. Artif. Intell.* 21(5):698-709.
- Vidal, V., and Geffner, H. 2006. Branching and pruning: an optimal temporal pool planner based on constraint programming. *Artif. Intell.* 170(3):298-335.