Common Subexpression Elimination in Automated Constraint Modelling

Ian P. Gent and **Ian Miguel** and **Andrea Rendl**[1]

Abstract. Typically, there are many alternative models of a given problem as a constraint satisfaction problem, and formulating an effective model requires a great deal of expertise. To reduce this bottleneck, automated constraint modelling systems allow the abstract specification of a problem, which can then be refined automatically to a solver-independent modelling language. The final step is to *tailor* the model to a particular constraint solver. We show that we can eliminate common subexpressions in the tailoring step, as compilers do when compiling source code. We show that common subexpression elimination has two key benefits. First, it can lead to a dramatic reduction in the size of a constraint problem, to the extent that solving time is reduced by an order of magnitude when the number of nodes searched is the same. Second, it can lead to enhanced propagation and reduced search. The effect of this can be even more dramatic, leading to reductions in nodes searched and time taken by several orders of magnitude. Where the technique does not lead to improved search, we have not seen it cause a significant overhead. Therefore, we propose that common subexpression elimination is an important technique for constraint programming.

1 INTRODUCTION

Constraint solving of a combinatorial problem, such as timetabling or planning, proceeds in two phases. First, the problem is *modelled* as a set of decision variables and constraints that a solution must satisfy. The second phase consists of using a constraint solver to search for solutions to the model: assignments of values to decision variables satisfying all constraints. Modelling a large, complex problem using constraints does, however, require expert knowledge. Such experts are few in number, preventing widespread access to constraint solving. One important obstacle is the *modelling bottleneck*. Not only are there many possible models for a given problem, but the model chosen has a substantial effect on the efficiency of constraint solving, and selecting an effective model is difficult.

Recent work has addressed this problem by allowing the user to describe a problem at a high level in an abstract constraint specification language, such as ESRA [6], ESSENCE [7], or Zinc [3], *without* being forced to make detailed modelling decisions. An automated system, such as CONJURE [8] or Cadmium [18], transforms this specification into a concrete model. This step is similar to program compilation. A compiler and a modelling system both refine a high-level language to an intermediate representation that is flattened to a target machine. The difference lies in the processed data: compilers deal with a set of instructions, Constraint Modelling deals with a set of relations.

[1] University of St Andrews, UK, email: {ipg,ianm,andrea}@cs.st-and.ac.uk

Eliminating common subexpressions is a technique that has been used successfully in Compiler Construction [5]. The idea is to enhance a program by detecting common pieces of code: if two pieces are equivalent, one piece can be omitted. Hence a reduction in execution time and memory usage is achieved.

This paper shows that eliminating common subexpressions in the context of constraint modelling conveys two key benefits. First, it can produce a significant reduction in the size of a constraint model. Second, it can lead to improved constraint *propagation* (inferences made by the constraint solver), and therefore dramatically reduced search. We show experimentally that the first benefit can lead to an order of magnitude improvement in run time in a constraint solver, while the second benefit can give several orders of magnitude improvement.

2 BACKGROUND

A constraint model is defined by a finite set of decision variables and a finite set of constraints on those variables. A decision variable represents a choice that must be made in order to solve the problem. The finite *domain* of potential values associated with each decision variable corresponds to the various options for that choice. A good model may be solved quickly while a bad model might not be solvable in a practical amount of time. An efficient model exploits both the modelling language's features and the constraint solver's strengths. Hence choosing an efficient representation requires a lot of expertise. Automated modelling seeks to ease this burden by automating as much as possible of the modelling process.

2.1 Automated Constraint Modelling

A number of approaches have been taken to automated constraint modelling. For example, the CONACQ [4] system uses machine learning to formulate a model from a set of solutions and non-solutions provided by the user. The O'CASEY system [15] uses case-based reasoning to store, retrieve and reuse constraint programming experience. In this paper our focus is on automated modelling through *refinement* of an abstract specification. In particular, we will discuss our approach in the context of the ESSENCE / CONJURE system, but it is equally applicable to similar systems such as Zinc / Cadmium. Indeed, common subexpression elimination could be used as a post-processing step to improve an existing model.

The ESSENCE language allows the specification of a problem *abstractly*, i.e. without making modelling decisions. ESSENCE, like Zinc, provides decision variables whose domain elements are combinatorial objects, such as sets, functions, or relations. Furthermore, these objects can be nested so that an individual variable may represent a set of sets, a set of sets of relations, and so on. This specifica-

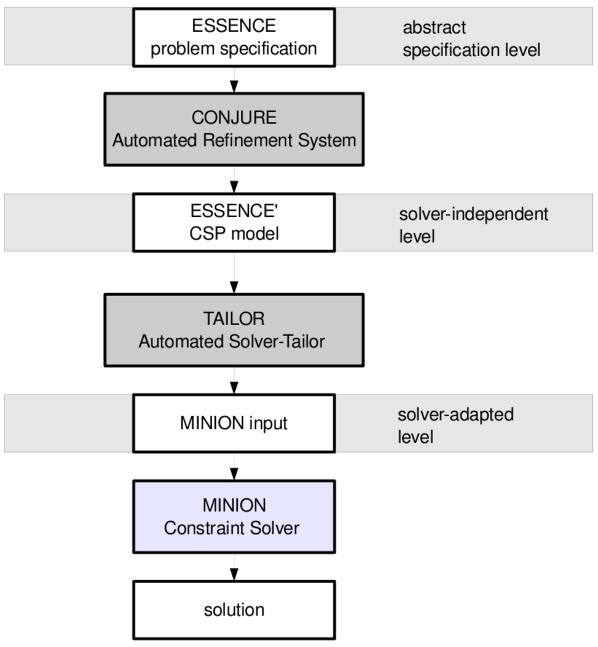

Figure 1. Automated Constraint Modelling with ESSENCE and CONJURE.

tion is refined automatically by the CONJURE system to a solver-independent constraint modelling language ESSENCE′. ESSENCE′ is a version of ESSENCE that has abstraction removed (principally, domain values are atomic) and provides facilities common to existing constraint solvers and toolkits. An ESSENCE′ model is adapted, or 'tailored' to a particular constraint solver using the TAILOR system [12]. The tailoring task consists of mapping ESSENCE′ expressions to the set of constraints provided by the target solver, as explained below. In this paper we will use MINION [11] as our target solver. An overview of the automated modelling process is given in Figure 1. At present, we are eliminating common subexpressions in the tailoring stage. In future, we plan to lift this work to the refinement stage.

3 TAILORING CONSTRAINT INSTANCES TO SOLVERS

Our approach focusses on tailoring problem instances rather than problem classes to a target solver. An instance is obtained from a class by giving a value for each parameter in the model (e.g. by giving $n = 8$, we obtain the 8-queens instance). Hence, each occurrence of a parameter in the model is instantiated to its associated value, which promotes tailoring steps such as evaluation and flattening. In our implementation, TAILOR adapts solver-independent constraint models to a particular solver by the following steps:

- Insert parameter values to obtain an individual instance
- Normalise the problem instance
- Flatten ESSENCE′ constraints (and variable datastructures) to conform those provided by the target solver
- Map flat, normalised instance to the target solver

Some constraint solvers, such as Minion, require individual instances as input. For others, our conjecture is that tailoring instances is worthwhile because of the extra information provided by instantiating the parameters. In the following we discuss the tailoring steps that are crucial for detecting common subexpressions efficiently.

3.1 Normalisation: A Prelude to Common Subexpression Detection

Our normalisation of ESSENCE′ has two components: evaluation and ordering. These are applied in an interleaved manner until a fixpoint is reached (the normal form). They are described below. We normalise expressions not only after parsing but also during flattening, when we unroll more complex expressions, such as quantifications. We do not apply any kind of factorisation of expressions.

3.1.1 Evaluation

Evaluation is particularly powerful when tailoring a problem instance to a target solver. Care is necessary in deciding the extent of evaluation: in some cases further evaluation might improve an instance but increase tailoring time and hence impair the (combined) modelling and solving process. Therefore, the expression evaluation included in our normalisation is simple, and cheap to perform. We evaluate constant expressions and apply several simple algebraic transformations, such as algebraic identity or algebraic inverses. We give some examples below.

$3 * 4 - 2$	$\longrightarrow$	10	Constant Evaluation
$exp + 0$	$\longrightarrow$	exp	Algebraic Identity
$exp - exp$	$\longrightarrow$	0	Algebraic Inverse
$exp \wedge false$	$\longrightarrow$	$false$	Logic

3.1.2 Ordering

We define a total order $\leq_o$ over the expressions of ESSENCE′. An ESSENCE′ model is transformed into a minimal form with respect to this order. The order represents a hierarchy of expressions, based on their complexity: Expressions on the bottom of the hierachy are expensive ones, such as non-linear expressions. Constants are at the top of this order, followed by variables and arrays. Further down the order are constraint types such as equalities, disequalities, inequalities, and special constraints like 'all-different'. Linear expressions come before non-linear expressions.

$$a + b + c + d \quad \leq_o \quad a * b$$

Expressions of different type are ordered based on their position in the order. Expressions of the same type are ordered recursively; each type has self-comparison rule. An equality constraint, for example, is ordered by examining the left argument first, followed by the right. The base case is where two constants or two variables are compared. In the former case, the comparison is by value, with least first. In the latter, the comparison is by name and domain. To illustrate, consider the following normalisations:

$$\begin{array}{lcl} x_4 + x_3 \neq x_2 + x_1 & \longrightarrow & x_1 + x_2 \neq x_3 + x_4 \\ x_5 * x_6 = x_8 + x_7 & \longrightarrow & x_7 + x_8 = x_5 * x_6 \end{array}$$

3.2 Flattening

It is common for constraint languages to support complex constraint expressions by re-writing, or *flattening*, them into a conjunction of simple constraints. In general, this mechanism is straightforward: replace a complex subexpression by an auxiliary variable that represents the subexpression. For example, an arithmetic expression, such as $a * (b + c)$, is flattened by replacing $b + c$ by an integer auxiliary variable aux_i and introducing an additional constraint $aux_i = b + c$. A relational expression, such as $a \Rightarrow (b \wedge c)$, is flattened by replacing $b \wedge c$ with a Boolean auxiliary variable aux_b. The equivalence between aux_b and $b \wedge c$ is expressed by a so-called *reification* constraint *reify*$(b \wedge c, aux_b)$ that corresponds to $aux_b \Leftrightarrow b \wedge c$ (aux_b is true if and only if the reified constraint $b \wedge c$ is satisfied).

	Unflattened	Flattened
Arithmetic Expression	$a * (b + c)$	$aux_i = a + b$ $a{*}aux_i$
Relational Expression	$a \Rightarrow (b \wedge c)$	*reify*$(b \wedge c, aux_b)$ $a \Rightarrow aux_b$

More complex expressions require several flattening steps to flatten completely, each introducing another auxiliary variable and equality constraint or reified constraint, respectively. As an example of a more complex expression, consider the constraint describing the 'legal moves' in the action-based model of the Peg Solitaire problem from [14] in Table 1.

4 COMMON SUBEXPRESSION ELIMINATION

This section discusses different sources of common subexpressions, together with efficient ways of detecting and exploiting them.

We say that two expressions are *common* (or equivalent) if they take the same value under all possible satisfying assignments. We distinguish between two types of equivalent subexpressions: subexpressions that are *syntactically* equivalent and subexpressions that are *semantically* equivalent. Syntactically equivalent expressions are *written* in the same way, such as a pair of occurrences of $a * b$. Semantically equivalent expressions *mean* the same thing, which can be deduced by their operational semantics, such as from the equivalence relation $a * b = c$. Clearly, syntactically equivalent expressions are also semantically equivalent.

Due to properties such as commutativity, many semantically equivalent logical and arithmetic expressions can be written so as to be syntactically distinct. A simple example is $a + b$ versus $b + a$. By normalising a constraint model (as we describe in Section 3.1) prior to common subexpression detection, the test for semantic equivalence is, in many cases, therefore reduced to the much cheaper test for syntactic equivalence.

The process of common subexpression elimination is straightforward. We record every two expressions that we denote equivalent. According to our expression ordering (see Section 3.1.2) the smaller expression is stored in a hashmap as representative for the greater expression. Whenever the greater expression re-occurrs in the constraint model, it is replaced by the smaller expression. Since the ordering captures the complexity of expressions, a subexpression is always replaced by a more effective subexpression.

Most common subexpressions arise in quantified expressions that are unrolled during flattening. Although a constraints expert can, of course, recognise common subexpressions and perform elimination manually, it is likely that a non-expert would not. Furthermore, even for an expert, performing this step in a complex model can be laborious and, without care, a source of error.

4.1 Explicit Common Subexpressions

A very simple example of common subexpressions is when the model contains a constraint of the form $X = Y$. We shall refer to constraints of this form as *explicit* equivalences, since they are given directly in the model.

4.1.1 Equivalence between Atomic Expressions

The simplest case of explicit common subexpressions $x = y$ is where x and y are atomic expressions, i.e. variables or constant values. The standard enhancement approach is to use x for every occurrence of y and, if y is a variable, to remove y from the set of variables, thus saving a variable. This approach has been extensively studied [1, 2, 16].

4.1.2 Equivalence between Compound Expressions

The general case of explicit common subexpressions $X = Y$ occurs when X and Y are arbitraily complex expressions. As example, consider the expression $x = y * z$. According to our ordering, x is cheaper than $y * z$ and we can replace every further occurrence of $y * z$ with x. Though exploiting equivalence between compound expressions can yield very effective results, this case mostly occurs in models formulated by non-experts.

4.2 Common Subexpressions Introduced During Flattening

The flattening process, which was explained in Section 3.2, naturally introduces a large number of equalities and reification constraints, which are a rich source of common subexpressions. If a certain subexpression appears again, we can simply *re-use* the auxiliary variable that already represents the subexpression, as the simple example below demonstrates:

Unflattened	Standard Flattening	Enhanced Flattening
$a + x * y = 0$ $x * y + b = t$	$aux_1 = x * y$ $a{+}aux_1 = 0$ $aux_2 = x * y$ $aux_2 + b = t$	$aux_1 = x * y$ $a{+}aux_1 = 0$ $aux_1 + b = t$

In our implementation, we flatten expressions bottom-up, i.e. expressions are flattened starting from the leaves of the expression tree. We maintain a hashmap that maps all previously flattened subexpressions to their corresponding auxiliary variables. Whenever we flatten a new subexpression we look up the hashmap for an equivalent expression: if we find an equivalent expression, we replace it with the respective auxiliary variable. This approach reduces the time required to match subexpressions and the memory we spend to collect previously flattened subexpressions. Note the importance of normalisation here: it is much easier to detect equivalence of normalised subexpressions.

The benefits we gain are great. First, if an instance contains common subexpressions of this kind, we save a variable and a set of constraints (depending on the complexity of the common subexpression) for every subexpression. Below, we report results to show that this effect on its own can reduce solving time by an order of magnitude, even without reducing the numbers of nodes searched.

We obtain a second large benefit, with even greater potential. This is that we can get additional propagation through re-using auxiliary variables. To see how this could happen, consider again the example above. Suppose that the domains of x and y are both $\{1, 2\}$. The domain of a is therefore $\{-4, -2, -1\}$ because $a + x * y = 0$. During search, we might set $b = 0, t = 2$. From this we can deduce $x * y = 2$ and in the standard flattening we get $aux_2 = 2$. However, we can deduce nothing about x or y because either $x = 1, y = 2$ or $x = 2, y = 1$ is possible, so nothing propagates through to aux_1 or a. When we use enhanced flattening, we share the same variable, so we deduce $aux_1 = 2$ and immediately propagate to set $a = -2$. Of course this can propagate further, depending on the problem. Thus, the simple detection of common subexpressions can lead to reduced search. Not only can it do this in principle, we will see below that it can reduce search by a factor of more than 2,000 in practice.

Some solvers flatten their input themselves, such as the Eclipse Constraint Programming System [10] which does not eliminate common subexpressions [19]. However, most solvers, such as MINION or Gecode [9] take a flattened model as input, hence flattening (in combination with common subexpression elimination) has to be done by the modeller - a tedious task, even for an expert (consider eliminating all common subexpressions of the 'legal moves'-constraint of the Peg Solitaire model in Table 1!). Therefore both Constraint novices and experts benefit from automated common subexpression elimination: poor models are drastically improved and good models might be improved if they contain common subexpressions without increasing tailoring time significantly.

***Peg Solitaire* Action model description**
We represent the board by *bState*, a list of squares for each step of the game. Every possible peq-move is assigned to a number between 1 and 76 and the array of variables *moves* holds the corresponding move for each step.

```
 0 given     noSteps : int
 1 letting   transitionStep :
 2             matrix indexed by [int(1..76),int(1..3)] of int(1..33) be ...
 3 letting   transitionNumber :
 4             matrix indexed by [int(1..33),int(1..33)] of int(0..76) be ...
 5 letting   STEPS be domain int(0..noSteps)
 6 letting   FIELDS be domain int(1..33)
 7
 8 find      bState : matrix indexed by [ STEPS, FIELDS ] of bool
 9 find      moves : matrix indexed by [int(0..noSteps-1) ] of int(1..76)
10
11  such that
12    . . .
13   $ legal moves
14    forall step : int(0..noSteps-1) .
15       forall f1,f2 : FIELDS .
16
17          $ if there exists a legal move from f1 to f2
18          (transitionNumber[f1,f2] != 0) ⇒
19
20          $ and we make that transition, the following holds..
21          ( (moves[step] = transitionNumber[f1,f2]) ⇔
22
23             ( (bState[step, f1] > bState[step+1,f1]) /\
24
25               (bState[step,transitionStep[transitionNumber[f1,f2],2]] >
26                bState[step+1, transitionStep[transitionNumber[f1,f2],2]]) /\
27
28               (bState[step, f2] < bState[step+1, f2]) /\
29
30               forall field : FIELDS .
31                 ( (field != f1) /\
32                   (field != transitionStep[transitionNumber[f1,f2],2]) /\
33                   (field != f2)
34                 ) ⇒
35                   (bState[step,field] = bState[step+1,field])
36               )
37          )
```

Table 1. Segment of the Peg Solitaire *Action* model [14] formulated in modelling language ESSENCE$'$. A summary of the model is given in Table 2

4.2.1 Example: Common Subexpressions in Peg Solitaire

As an example for common subexpressions, consider the partial model of the Peg Solitaire Problem [14] in Table 1. Peg Solitaire is a gamed played on a board with holes and pegs to arrange. The aim in the standard version of the game is to perform checkers-like moves to remove all pegs but one from the board.

We represent the 33 fields (holes) on the board by booleans: *true* states that a peg is in the hole and *false* states that the hole is empty. The board changes after every move, so we represent the board states by the matrix *bState*, where the ith vector represents the field-variables for the ith step in the game. There are 76 possible moves on the board and the 1-dimensional matrix *moves* holds the variables for each move made in the game. The constant matrix *transitionNumber*$[f_1,f_2]$ gives the corresponding transition number (ranging from 1 to 76) when making a move from field f_1 to f_2. *transitionStep* [*step*, *i*] gives the field-variable that is involved when performing the step with number *step*.

We constrain the move chosen to be legal using a universal quantification in line 14. A summary of the 'legal moves'-constraint is given in Table 2. Recall that such quantified 'loops' must be unrolled for constraint solvers such as Minion or Gecode. As we do so, common subexpressions arise between the expressions obtained for different values of the quantified variable. An example is the inequality in line 23 that is nested in a conjunction,

$$bState[step, f1] > bState[step + 1, f1]$$

When the quantification is unrolled for *step*=0 and f_1=2, it yields the subexpression

$$bState[0, 2] > bState[1, 2]$$

that re-occurs every time field 2 is involved in another move at step 0. The same holds for the other inequalities from lines 25 and 28.

Summary of action-centric model of Peg Solitaire where *move* is an array of variables representing the moves required to solve the puzzle, *bState* is an array of variables representing the state of the board at each step, t ranges over the steps in the sequence of moves, m is a move, start(m), mid(m) and end(m) return the three positions affected by move m, and unchanged(m) returns the set of positions *not* affected by move m.

```
forall $t$ in 1..31 .
forall $m$ in 1..76 .
    $move[t] = m \leftrightarrow$
        $bState[t-1, \text{start}(m)] > bState[t, \text{start}(m)] \wedge$
        $bState[t-1, \text{mid}(m)] > bState[t, \text{mid}(m)] \wedge$
        $bState[t-1, \text{end}(m)] < bState[t, \text{end}(m)] \wedge$
        forall $u$ in unchanged(m) . $bState[t-1, u] = bState[t, u]$
```

Table 2. Summary of action-centric model of Peg Solitaire

5 EXPERIMENTAL RESULTS

In this section we compare models that we tailor (as described in Section 3) either with or without common subexpression elimination. We present a selection of problems that we formulate in ESSENCE$'$ without applying symmetry breaking. Then we tailor the ESSENCE$'$ model to a MINION instance using the tool TAILOR, in which we have implemented (optional) common subexpression elimination. For each problem instance, we generate two different MINION input files: one that is tailored by eliminating common subexpressions and one that is not. Both models are solved on the same machine (Dual-core Intel P4 at 3GHz with 1.5Gb RAM) using MINION v0.5. We apply the same variable ordering heuristic (decision variables first,

n	Tailoring (s)		Solving Time (s)		Search Nodes		Common	Aux Variables		Constraints	
	♠	♡	♠	♡	♠	♡	Subexpr.	♠	♡	♠	♡
5	0.36	0.37	9.49	0.04	400,399	1870	1,230	1,440	200	1,486	256
6	0.41	0.42	1809.53	0.39	79,159,269	32,964	2,248	2,535	287	2,614	366
7	0.53	0.49	48,020.50	8.58	1,448,334,418	604,206	3,710	4,101	391	4,206	496

Table 3. Solving performance and model features of Peaceable Army of Queens models with (♡) and without(♠) common subexpression elimination

then auxiliary variables) and same value ordering heuristic (ascending) in both cases. We compare the models in several ways. As well as solving performance we report tailoring time. We also look at features of the tailored instances, such as the number of constraints and auxiliary variables.

5.1 Golomb Ruler

The Golomb Ruler problem is to find a ruler of minimal length with n ticks such that the distance between every two ticks is different. Our results with common subexpression elimination are very interesting: we take the basic model from [13] which uses quarternary constraints to express the distances between the ticks. Applying common subexpression elimination on the basic model automatically yields the enhanced distance model from [13]. Hence this example demonstrates how weak models can automatically be enhanced to advanced, effective models from the literature. The results are given in Table 4: we gain a great reduction in search time but also in search space.

	Common	Solving Time		Search Nodes	
n	Subexpr.	♠	♡	♠	♡
7	679	0.97	0.04	2,507	1,996
8	1260	26.80	0.35	22,508	17,427
9	2148	513.54	3.27	188,026	141,503
10	3435	>7,461.07	36.32	>1,406,328	1,114,964

Table 4. Solving performance of Golomb Ruler instances with(♡) and without(♠) common subexpression elimination

5.2 Peg Solitaire

We formulated two different models of Peg Solitaire. The first is taken from [14] and is *state*-centric: for each possible change to the state of the board, a constraint is added specifying the moves that might be responsible. We also experimented with a novel *action*-centric model: a constraint is added for each possible action, specifying the changed and unchanged parts of the board. The constraints are briefly summarised in Table 2.

In the action-centric model, we reduce the number of auxiliary variables from 87,172 to 6,603, and the number of constraints from 89,625 to just over or under 9,000 depending on the starting position. In the state-centric model, we reduce the number of auxiliary variables from 313,720 to 12,989 and the number of constraints from 316,886 to 16,155. We present solving results in the two models (from three different starting positions) in Table 5. In the action-centric model, we get no reduction in search nodes, a small increase in tailoring time, but an order of magnitude reduction in run time. In the state-centric model, we do in fact get a reduction in search of about a factor of 3, as well as a reduction in time taken per node. Note that the elimination of common subexpressions reverses the performance of models. That is, the action-centric model is best without subexpression elimination, but when it is used the state-centric model searches faster. When tailoring time is also taken into account, the action-centric model is fastest overall on the easiest instance, but state-centric remains best for the two harder instances.

start	Tailoring (s)		Solving (s)		Search Nodes	
field	♠	♡	♠	♡	♠	♡
17	5.88	5.97	31.7	3.2	10,269	10,269
10	5.87	6.18	4376.2	456.7	1,486,641	1,486,641
5	5.97	6.04	>7200	3,920.4	11,398,210	11,398,210
17	47.06	44.37	42.4	2.7	10,269	3,944
10	46.55	44.24	6,383.2	247.4	1,486,641	539,374
5	46.64	44.70	>7200	2,151.8	>1,784,832	3,066,971

Table 5. Action-centric (top) and state-centric (bottom) Peg Solitaire models with (♡) and without (♠) common subexpression elimination

5.3 Peaceable Army of Queens

The peaceable army of n queens problem is to place two equally-sized armies of white and black queens on an $n \times n$ chessboard such that no queen can attack a queen of the other colour. We formulate the 'basic model' of Smith *et al* [17] without symmetry breaking constraints in ESSENCE$'$. We compare performance (to find an optimal solution and prove its optimality) and the models in Table 3. Common subexpression elimination has a more dramatic impact than in the previous experiment. Here, we see the number of search nodes reduced from 1.5 billion to less than a million at $n = 7$, a factor of more than 2,000. Solving time is reduced even more, by more than 5,000 times at $n = 7$. These dramatic improvements occur through improved propagation after we have eliminated common subexpressions. They allow reasoning to occur over parts of the model which are separated in the vanilla model. Comparison with results of [17] is inconclusive. Our results without common subexpression detection are much worse than reported there, while results with it are similar. We do not know if this is because some feature of our model which is different in detail to theirs, or Minion propagates the same model worse, or whether Smith *et al* may have eliminated subexpressions without detailing it. None of these explanations would invalidate our main point, that common subexpression elimination can, on its own, make a very bad model much better purely automatically.

5.4 Balanced Incomplete Block Design (BIBD)

BIBD is problem 28 in CSPLib [20]. We use the standard model from the literature, consisting of 0/1 variables, sums and scalar products. The model does not contain common subexpressions so we cannot improve the model during flattening. Still, we generate two MINION models from each instance: one where we try to eliminate common subexpressions (in vain) and one without. We don't give a model comparison since the generated models are identical, but investigate tailoring and solving time in Table 6. This comparison is very interesting: we observe that we do not suffer significantly from the attempt to eliminate common subexpressions, even though there are none. Translation times are no more than 30% higher when failing to find any common subexpressions. We generate an identical model and get identical search results in terms of nodes searched, with very similar search times. Fluctuations in search time are presumably just the difference between separate runs. From this experiment we draw the conclusion that the attempt to eliminate common subexpressions - even in vain - does not significantly slow down the modelling and solving process.

b, v, r, k, λ	Tailoring (s)		Solving Time (s)		Search Nodes	
	♠	♡	♠	♡	♠	♡
7,7,3,3,1	0.28	0.24	0.01	0.01	21	21
140,7,60,3,20	0.51	0.59	0.43	0.44	17,235	17,235
210,7,90,3,30	0.68	0.82	2.61	2.63	67,040	67,040
280, 7,120,3,40	0.90	1.15	9.92	9.51	182,970	182,970
315,7,135,3,45	1.04	1.26	16.05	17.05	278,310	278,310
385, 7,165,3,55	1.29	1.64	44.17	44.30	574,365	574,365

Table 6. Solving performance of BIBD models with (♡) and without(♠) common subexpression elimination. No common subexpressions were found.

6 RELATED WORK

Le Provost and Wallace discuss derivation and elimination of common subexpression during propagation in [2] but restrict their discussion to explicit atomic subexpressions. Harvey and Stuckey eliminate explicit atomic and linear subexpressions in their work on improving linear constraint representations in [1]. Neither study addresses common subexpression elimination during flattening nor elimination of non-linear constraints, as we do in our work.

In their work on interval analysis, Schichl *et al* [21, 22] discuss common subexpression elimination in models of mathematical problems represented as directed acyclic graphs. These studies have much in common with our work, and further examine the issue of propagation over common subexpressions. However, they do not include logical expressions, such as quantification, which we have identified as one of the main sources of common subexpressions.

7 CONCLUSIONS

We have shown that common subexpression detection, common in compilers, can be applied successfully to constraint modelling. We have shown that this can be implemented effectively as part of the TAILOR system, which translates models from a solver-independent modelling language to a target constraint solver.

Our experimental results show three things. First, we can obtain an order of magnitude improvement in run time simply by reducing the number of variables and constraints, with no change in search space. Second, we can obtain additional propagation, resulting in orders of magnitude improvements in the search space and run time. Third, although these improvements are not always possible, we do not pay a significant penalty where we cannot find common subexpressions. Taken together, the huge benefits outweigh the low costs, and common subexpression elimination should be considered wherever possible.

It could be argued that tailoring a simple model without common subexpression detection is a straw man, because other models perform better. There may be other models which are inherently better, with or without common subexpression detection. It is also true that any model reduction achieved automatically can also be achieved manually. However, these potential criticisms do not address the true point of our work. First, if we can improve a poor model by a factor of thousands in run time, we may avoid the need to spend time thinking of a better model. The result may be much better in terms of time required of an expert constraint modeller. Second, while common subexpression detection *could* be done by humans, it is *not* done in practice. In fact, it would often be impractical. A modeller would have to study a model looking for repeated expressions, and then remodel manually while avoiding mistakes in doing so. It is preferable to automate this process. Hence both modelling expert and novice benefit from automated common subexpression elimination.

Acknowledgements Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. Andrea Rendl is supported by a DOC fFORTE scholarship from the Austrian Academy of Science and UK EPSRC grant EP/D030145/1. We thank Warwick Harvey and our anonymous reviewers for their helpful comments.

REFERENCES

[1] W. Harvey, P. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation, *Constraints*, 7, pp172–203, 2003.
[2] T. Le Provost, M. Wallace Generalized Constraint Propagation over the CLP Scheme. J. Logic Programming, 16(3), 1993.
[3] M. de la Banda, K. Marriott, R. Rafeh, M. Wallace. The modelling language Zinc. *CP*, 700-705, 2006.
[4] C. Bessiere, R. Coletta, F. Koriche, B. O'Sullivan. Acquiring Constraint Networks using a SAT-based Version Space Algorithm. In *AAAI*,, pp 1565-1568, 2006.
[5] J. Cocke, Global common subexpression elimination, *SIGPLAN Not.*, 5:20–24, 1970.
[6] P. Flener, J. Pearson, M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. *LOPSTR '03: Revised Selected Papers*, LNCS 3018, 2004.
[7] A.M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of Essence: A constraint language for specifying combinatorial problems. *IJCAI*, pp80-87, 2007.
[8] A. Frisch, C. Jefferson, B. Martínez Hernández, and I. Miguel. The rules of constraint modelling. *IJCAI*, pp 109–116, 2005.
[9] Gecode: a Generic Constraint Development Environment http://www.gecode.org
[10] The ECLiPSe Constraint Programming System http://eclipse.crosscoreop.com/
[11] I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. *ECAI*, pp 98–102, 2006.
[12] I.P. Gent, I. Miguel and A. Rendl. Tailoring Solver-independent Constraint Models: A Case Study with Essence′ and Minion In *SARA*, pp 184-199, 2007.
[13] B. M Smith, K. Stergiou and T. Walsh. Modelling the Golomb Ruler Problem School of Computing Research Report 1999.12, University of Leeds, June 1999.
[14] C. Jefferson, A. Miguel, I. Miguel, A. Tarim. Modelling and Solving English Peg Solitaire. In *Computers and Operations Research* 33(10), pages 2935-2959, 2006.
[15] J. Little, C. Gebruers, D. G. Bridge and E. C. Freuder, Using Case-Based Reasoning to Write Constraint Programs. In *CP*, pp 983, 2003.
[16] B.A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n Queens. *IEEE Expert* 5:16-23, 1990
[17] B.M. Smith, K.E. Petrie, and I.P. Gent. Models and symmetry breaking for peaceable armies of queens. In *Proceedings CPAIOR 04*, pages 271–286, 2004.
[18] P. J. Stuckey, M. de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, T. Walsh. The G12 Project: Mapping Solver Independent Models to Efficient Solutions. *CP*, 13-16, 2005.
[19] Warwick Harvey Personal Communication, May 2008
[20] CSPlib: A Library for Constraint Problems http://www.csplib.org/
[21] H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization Journal of Global Optimization 33/4 (2005), 541-562
[22] X.-H. Vu, H. Schichl and D. Sam-Haroud Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems In *ICTAI 2004*, 72-81