

Automated Constraint Model Enhancement during Tailoring

Andrea Rendl, Ian Miguel

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

Abstract. ¹Constraint modelling is difficult, particularly for novices. Hence, automated methods for improving models are valuable. The context of this paper is *tailoring*, a process where a solver-independent constraint model is adapted to a target solver. Tailoring is augmented with automated enhancement techniques, in particular common subexpression detection and elimination, which, while powerful, can be performed inexpensively if applied selectively. Experimental results show very substantial improvements in search performance of tailored models.

1 Introduction

Modelling is a major bottleneck in the process of solving a problem using constraints. Many models are possible for a given problem, and the model chosen has a substantial effect on the efficiency of the solving process. However, it is difficult (especially for a non-expert) to know which is the best model to choose. Therefore, any automated means by which a given model can be improved automatically is valuable.

The context of this work is *tailoring* [9], a process where a solver-independent constraint model is adapted to a target solver. This is an important step, since different constraint solvers have different strengths, weaknesses and facilities, such as the library of decision variable types and constraints. In general, tailoring involves adapting constraints, variable types and stated heuristics to the target solver's repertoire, which includes flattening of constraints, propagator selection and adaption of heuristics. Most solver-independent modelling languages require some form of tailoring.

We investigate augmenting tailoring by a set of enhancement techniques, many of them successfully applied in related fields, such as Compiler Optimisation, Satisfiability and Proof Theory. Crucially, we show how these techniques can be applied in the context of constraint modelling for little additional computational effort. Our primary focus is on the efficient detection and exploitation of common subexpressions, both syntactic and semantic. We begin by discussing the enhancement of individual problem instances, then describe ongoing work in lifting this approach to entire problem classes. Furthermore, we show how removing common subexpressions can enable further useful model enhancements. Our experimental results demonstrate the efficiency of our approach.

¹ An earlier version of this paper was submitted to the Symposium on Abstraction, Reformulation and Approximation. This is in accordance with the CFPs of both SARA (which does not require copyright) and CP (which explicitly allows submissions of papers previously submitted to events of lower status than a conference).

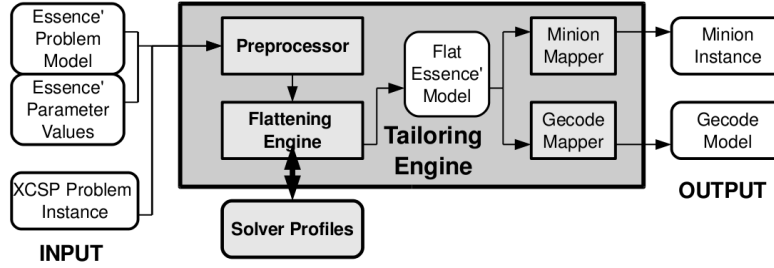


Fig. 1. General Tailoring Approach in TAILORv0.3.2. As input, TAILOR either takes a problem class and an parameter file (both formulated in ESSENCE'), or an instance formulated in XCSP format 2.1 [25]. First, the input is preprocessed, then flattened using solver profiles. The resulting flat model can either be mapped to target solver Minion or Gecode.

2 Tailoring and Solver Profiles

There exist several solver-independent constraint modelling languages, such as OPL [13], MINIZINC [18] or ESSENCE' [9]. Herein, all constraint expressions are formulated in ESSENCE' — but this choice is unimportant. Typically, a constraint model represents a parameterised problem *class* (e.g. the n -queens problem). A problem *instance* is obtained by instantiating a problem class with parameters, e.g. the 4-queens instance is obtained from the n -queens problem class where parameter n is set to 4. A problem class formulation is usually paired with a parameter file to create an instance.

We augment TAILOR [9] (Fig. 1). TAILOR takes two kinds of input: either an ESSENCE' problem model (optionally paired with a parameter file), or an XCSP 2.1 [25] instance. First, the input is *preprocessed* (parameters/constants insertion, normalisation) then *flattened*, resulting in a flat ESSENCE' representation (similar to FlatZinc [18]), which is then mapped to the solver representation. Many constraint solvers exist, each with its own strengths and weaknesses, each tackling problem instances in its own way. Problems on which some solvers struggle can easily be solved by others and vice versa [3]. Therefore, it is advantageous to target a number of solvers [23]. Currently, TAILOR targets solvers MINION [8] and Gecode [7]. For brevity, we focus on MINION herein.

Flattening decomposes a constraint expression into an equivalent conjunction of simpler expressions, replacing a subexpression with an auxiliary variable. This is necessary when the target constraint solver does not support the original expression. A main challenge of generalised tailoring is the generalisation of the flattening procedure: every constraint solver has a different library of constraints, hence many expressions are flattened differently for each solver. As an example, consider flattening the nonlinear constraint ' $a + b + c \neq e * f$ ' to three different solvers: Eclipse [5], Gecode and MINION. The appropriate constraint representation for each solver is:

Eclipse Prolog	Gecode	MINION
$a + b + c \neq e * f$	$aux_1 = e * f$ $a + b + c \neq aux_1$	$aux_1 = e * f$ $a + b + c \leq aux_2$ $a + b + c \geq aux_2$ $aux_1 \neq aux_2$

Prolog takes arbitrarily complex expressions, hence no flattening is required. Gecode provides a linear disequality constraint, allowing variables as arguments only, hence $e * f$ is flattened by introducing auxiliary variable aux_1 and posting $a + b + c \neq aux_1$. MINION only provides binary disequality, hence we introduce another auxiliary variable, aux_2 ,

representing $a + b + c$. Note, that the flat representation of MINION would also be valid for solvers Gecode and Eclipse Prolog, but would contain additional variables, aux_1 and aux_2 , respectively. Such a representation can result in worse propagation/runtime than a representation that is exactly tailored to the solver’s repertory. Therefore, the flattening engine should decompose expressions *only* if the expression is not directly supported by the solver. Hence, the flattening engine requires information about the solver’s constraint repertory - information a *solver profile* can provide.

A solver profile captures important features of a particular solver, such as variable information (variable types, data structures), propagator information (constraint type, consistency level, arity, reifiability, etc.), search heuristics and other, solver-specific features. Given a general list of features, every solver profile associates a boolean value to each feature that indicates if it is supported or not. This information can be used to guide flattening, similar to rule-based systems in retargetable compilers [6]. When given an expression the flattening engine consults the solver profile about the availability of the corresponding propagator. If no applicable propagator exists, flattening proceeds (e.g. the n-ary multiplication is flattened into a binary multiplication). In this manner, an expression is flattened only if the target solver does not support it.

This approach provides three key benefits: first, it assists in reducing the overhead when flattening expressions, since expressions are only flattened if necessary for the target solver. Second, a general flattening engine can be used for different solvers. Third, the flexibility of the solver profile allows to easily adapt to changes in the target solver (e.g. a new constraint is supported by simply changing the settings in the solver profile). Solver profiles can also assist in other parts of the tailoring process, such as selecting an appropriate propagator or search heuristic (in case favoured propagators or search heuristics are not already defined in the modelling language, as possible in MINIZINC).

3 Common Subexpressions in Constraint Models

Two expressions are *common* (or equivalent) if they take the same value under all possible satisfying assignments. There are two types of equivalent subexpressions: *syntactically* equivalent and *semantically* equivalent subexpressions. 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, e.g. expressions $a*b$ and $b*a$ are equivalent. Syntactically equivalent expressions are also semantically equivalent, however, in this paper, we will refer to semantically equivalent expressions as those that are *not* syntactically equivalent.

Novelty Common subexpression elimination is an optimisation technique originating from compiler optimisation [4], that has proven to be powerful in several related disciplines, such as Satisfiability [16], Model Checking [14], Proof Theory [19] and Numerical CSPs [1]. In their work on interval analysis, Schichl *et al* [20, 24] discuss common subexpression elimination in models of mathematical problems represented as directed acyclic graphs. These studies have much in common of with work, and examine the issue of propagation over common subexpressions. However, they do not cover logical expressions, such as quantification, an important source of common subexpressions in constraint problems, as we will show.

Exploiting explicit linear equalities has been well studied [12, 15, 17]. As an example, consider the explicit linear equality $x = y$, where x and y are decision variables.

If x and y have the same domain, every occurrence of y can be replaced with x (or vice-versa) and y removed from the set of variables. Otherwise, a new variable, ranging over the intersection of x and y 's domain, can replace both throughout.

This work is concerned with the advanced case, where the equivalence between two expressions is not explicitly given, as above, but has to be *detected*, either by checking for syntactic or semantic equivalence. We show how to exploit the tailoring process to integrate common subexpression detection and elimination in a computationally cheap way. The elimination of syntactic equivalences is discussed in Section 4, whereas semantic equivalence is covered in Section 5.

We want to stress that despite the obvious benefits of common subexpression elimination, it is *not* routinely performed by constraint solvers: solvers that expect a pre-flattened input, such as MINION or Gecode have no opportunity. Solvers that allow nested input, such as Eclipse or Choco [2], or tailoring tools like the MiniZinc-FlatZinc translator [18], do not make use of common subexpressions.

Common Subexpressions in Constraint Models Both syntactically and semantically common subexpressions occur naturally and often in constraint instances. Typically, unrolling quantifications exposes common subexpressions. Quantified expressions are *unrolled* when deriving an instance from a problem class by instantiating the problem parameters. As an example, consider the Golomb Ruler Problem that is concerned with finding a ruler of minimal length with n ticks where all distances between ticks are distinct. The distance constraint of a basic constraint model [21] is given below:

$$\begin{aligned} &\mathbf{forall} \ i, j, k, l : \text{int}(1..n) . \\ &((i < j) \wedge (k < l) \wedge ((j > l) \vee (i > k))) \Rightarrow \\ &(\text{ticks}[j] - \text{ticks}[i] \neq \text{ticks}[l] - \text{ticks}[k]) \end{aligned}$$

The 1-dimensional array *ticks* represents the position of each tick, and parameter n denotes the number of ticks. The constraint states that the distance between every pair of ticks must be different. When unrolling the quantification, we get the set of constraints

$$\begin{aligned} &\text{ticks}[3] - \text{ticks}[1] \neq \text{ticks}[2] - \text{ticks}[1] \\ &\text{ticks}[3] - \text{ticks}[2] \neq \text{ticks}[2] - \text{ticks}[1] \\ &\text{ticks}[3] - \text{ticks}[2] \neq \text{ticks}[3] - \text{ticks}[1] \end{aligned}$$

which contain several occurrences of each subexpression $\text{ticks}[2] - \text{ticks}[1]$, $\text{ticks}[3] - \text{ticks}[1]$ and $\text{ticks}[3] - \text{ticks}[2]$. Although a constraints expert can, of course, recognise common subexpressions and perform elimination manually, it is likely that a non-expert would not. Even for an expert, performing this step in a complex model can be laborious and, without care, a source of error.

4 Eliminating Syntactically Common Subexpressions

We propose the detection and elimination of common subexpressions during flattening. Flattening is a recursive process that, when given an expression, replaces its subexpressions with auxiliary variables, if appropriate for the target solver (see Section 2). Hence, in a typical flattening engine, two common subexpressions will be represented by two different auxiliary variables. However, if the flattening engine is able to detect the equivalence between two common subexpressions, it can replace both subexpressions with the same auxiliary variable, thus saving one variable.

In order to perform the detection step, we simply augment the flattening process to record each flattened subexpression together with its associated auxiliary variable in a

hashmap as it is introduced. Whenever we flatten a new subexpression, we test for a match in the hashmap (this is a test for *syntactic* equivalence). If an equivalent expression is found, we replace the subexpression with the existing auxiliary variable, rather than creating a new one. This approach reduces the time required to match subexpressions and the memory we spend to collect previously flattened subexpressions.

Embedding common subexpression detection and elimination in flattening in this way is particularly attractive because it produces a monotonic reduction in the number of constraints and variables in the model without adding significant computational overhead. Furthermore, it guarantees that we only eliminate common subexpressions that *need* to be flattened and therefore we do not impair the model. As an example, consider the two linear constraints: $x - y \leq a$ and $x - y \leq b$ that share the subexpression $x - y$. In our approach, we would not eliminate $x - y$ because linear constraints of this form generally do not require flattening. Nevertheless, we could eliminate $x - y$ by introducing an additional variable aux and post the constraints $aux = x - y$, $aux \leq a$ and $aux \leq b$. However, this representation has worse propagation than the initial two constraints, since eliminating $x - y$ introduces overhead (one additional variable and constraint) without reducing the complexity of the initial constraints ($aux \leq a$ and $aux \leq b$ are still ‘only’ linear). Linear propagators are very powerful and the number of arguments ($x - y \leq a$ or $aux \leq a$) does not matter greatly. This example highlights that common subexpression elimination should only be performed if the elimination does not introduce additional variables. Clearly, this premise holds during flattening, since common subexpressions are only eliminated if they have to be flattened to an auxiliary variable in the first place.

4.1 Benefits of Common Subexpression Elimination

The benefits we gain are great. First, if an instance contains common subexpressions of this kind, we save at least one variable and constraint (depending on the complexity of the common subexpression) for every subexpression. Second, we can reduce solving time by up to an order of magnitude, as we report in the Section 7. Less importantly, we can also reduce the flattening time, since we do not spend additional time on flattening expressions that we have already flattened. The third large benefit has even greater potential: we can get additional propagation through re-using auxiliary variables. We illustrate this by the example given in the table below.

Unflattened	Flattened with CSE.	Standard Flattening
$a+x*y=b$ $b+x*y=t$	$aux_1=x*y$ $a+aux_1=b$ $b+aux_1=t$	$aux_1=x*y$ $a+aux_1=b$ $aux_2=x*y$ $b+aux_2=t$

Suppose that the domains of x and y are both $\{1, 2\}$. 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. From $a+aux_1=0$ and $x, y \in \{1, 2\}$, propagation will only result in the domain of aux_1 set to $\{1, 2, 4\}$ and the domain of a set to $\{-4, -2, -1\}$. 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.

5 Eliminating Simple Semantic Equivalences

Some constraint models contain semantic equivalences that are not syntactically equivalent, and hence not eliminated using our approach described in Section 4. Detecting all semantically common subexpressions is an operation that can be arbitrarily hard (e.g. detecting the equivalence between a clique of disequalities and global constraint *alldifferent*). However, there are many *simple* semantic equivalences that can be easily detected and *reduced* to syntactic equivalence, and hence eliminated. In this section we first discuss how to reduce semantic equivalence to syntactic equivalence and then describe two different approaches of detecting semantic equivalences.

5.1 Reducing Semantic Equivalence to Syntactic Equivalence

Two semantically common subexpressions (that are not syntactically common) are reduced to two syntactically common subexpressions by re-writing one expression to the representation of the other (e.g. given $a*b$ and $b*a$, we re-write $b*a$ into $a*b$, gaining two occurrences of $a*b$). To perform this step, one of the two representations has to be chosen - preferably the most effective one for the constraint model. In the case of $a*b$ and $b*a$ this choice is easy (it makes no difference), but in many cases it is not. For instance, consider the semantically common subexpressions $a*(b+c)$ and $a*b+a*c$. Generally, the first representation is preferable, since it provides better propagation. However, if $a*b$ and $a*c$ have common subexpressions in the model and $b+c$ does not, then the second representation is to be preferred.

Clearly, some equivalence relations cannot be classified without considering the rest of the constraint model. We therefore distinguish between two kinds of equivalence relations: those where we can determine the preferable representation immediately (e.g. $a*b$ and $b*a$), and those that require further investigations. The detection of the first kind of equivalence relations, which is cheap to perform, is integrated into the preprocessing phase of tailoring. The detection of the second kind is integrated into the flattening procedure, where information about other, previously flattened subexpressions is available. Note that both approaches do not guarantee the detection of *all* semantically common subexpressions of this kind - a tradeoff for investing little computational effort into the procedures. We discuss both approaches in more detail below.

5.2 Detecting Semantic Equivalence during Preprocessing

The simplest case of semantic equivalences are those given by commutativity and associativity, e.g. $2*x$ and $x*2$, which can easily be reduced by normalisation. Normalisation is a wide-spread technique to transform expressions into a normal form. Our normalisation of ESSENCE' has two components, *evaluation* and *ordering*, that are applied in an interleaved manner until a fixpoint is reached.

Evaluation is important to simplify expressions involving constants and is performed only to a certain extent to minimise the computational effort. We evaluate constant and simple logical expressions and apply several simple algebraic transformations, such as algebraic identity or inverses. Note, that evaluation also reduces basic semantic equivalences, such as of $(7+4)*x$ and $11*x$.

The main reduction of semantic equivalent expressions results from ordering expressions. We define a total order over the expressions of ESSENCE' and transform each expression into a minimal form with respect to this order. The ordering affects commutative operators only. As an example, $b*a=c$ is ordered to $c=a*b$. Note, that ordering does not reveal all common subexpressions in commutative expressions. Consider the two subexpressions $a + b + c$ and $a + c$. Ordering will not detect that $a + b + c$ contains $a + c$. This is a tradeoff for investing little time into detection. Moreover, detecting $a + c$ in $a + b + c$ is relevant only, if linear sums are represented by ternary propagators in the target solver. However, most target solvers provide n -ary propagators for commutative operators (e.g. summation, disjunction, conjunction), so detecting these equivalences is mostly not necessary.

5.3 Detecting Semantic Equivalences during Flattening

Our approach of detecting semantically common subexpressions is embedded into the elimination process during flattening, as described in Section 4. The idea is to augment this detection process by the following step: whenever an expression that is to be flattened has *no* syntactically common subexpression, we reformulate it, using simple equivalence rules, and test the resulting expression and its subexpressions for syntactically common subexpressions.

The reformulations we investigate are very simple, thus easy and cheap to perform. We restrict the reformulations to a particular set of candidates only, to reduce the computational effort. Since we perform detection during flattening, the order of constraints has a great influence on which common subexpressions we detect. Hence, this approach of detecting semantically common subexpressions is *not* confluent.

Negation-Reformulation is concerned with transforming particular subexpressions to their negated form. For instance, consider the subexpression $A < B$ whose negated form would be $\neg(A \geq B)$. Whenever $A < B$ has no common subexpression in the model, we test its negated form, $\neg(A \geq B)$, and its subexpression $A \geq B$ for common subexpressions. We apply this strategy to expressions composed by relational operators (e.g. \neq, \geq, \dots), since they are most likely to contain equivalent subexpressions from our experience. For instance, consider $(A = B) \Rightarrow D$ and $(A \neq B) \Rightarrow C$. If $A \neq B$ has no common subexpression, we reformulate $A \neq B$ to $\neg(A=B)$ and detect the common subexpression $A=B$. Assume $A=B$ is represented by auxiliary variable aux , then $A \neq B$ is presented by $\neg aux$ instead of introducing a new auxiliary variable for $A \neq B$.

Detecting common subexpressions by this reformulation is clearly not confluent. If we switch the order of the two constraints in the example above, then the two subexpressions will be flattened the other way round: $A \neq B$ will be flattened to aux and $A=B$ to $\neg aux$. However, this representation impairs the model since the equivalence relation $A=B$, expressed by $\neg aux$, corresponds to $\neg\neg A=B$. Hence we only perform the *negation*-reformulation on disequality constraints and not vice versa.

Horn Clause-Reformulation A horn clause is a disjunction of literals where at least one literal is negative, i.e. the disjunction can be expressed as an implication. As an example, consider the expression $A \Rightarrow B$ where A and B are arbitrary relational expressions. Its horn clause representation is $\neg A \vee B$. If neither $A \Rightarrow B$, A nor B have a common subexpression, then the alternative representation, $\neg A \vee B$, or $\neg A$, can be tested for a common subexpression (and vice versa).

Other Simple Reformulations We can use De Morgan’s Law as reformulation to create and detect further common subexpressions. As an example, consider the expression $\neg \bigwedge_i^{1..n} E_i$ where E_i is an arbitrary expression, dependant on i . Using De Morgan’s law, it can be reformulated to $\bigvee_i^{1..n} \neg E_i$. This reformulation can be used to match $\neg E_i$ with a common subexpression. Similiarly, the law of distributivity can be exploited to create expressions that are likely to match other subexpressions.

6 Tailoring and Enhancing Problem Classes

In the previous sections, we have focussed on tailoring and enhancing problem *instances*, whereas now we extend our discussion to tailoring whole problem *classes*. There are two main reasons for tailoring problem classes: first, to support solvers that take problem classes as input: solvers such as Gecode, Eclipse or Choco are libraries of programming languages, where problems are formulated as programs and parameters can be specified at runtime. Second, since all enhancements on instance level are applicable to problem class level, it is more time-efficient to perform them *once* on problem class level instead of *several times*, for every problem instance.

For brevity, we will restrict our discussion to a single setup: flattening ESSENCE’ problem classes to solver MINION, which results in a flat, enhanced ESSENCE’ problem class. This flat representation can then be used for tailoring instances, which can save essential tailoring time. However, tailoring problem classes is more challenging than tailoring instances. In this section, we discuss the differences between flattening instances and problem classes and extend the discussion of detection and elimination common subexpressions to problem class level. All features discussed in this section are implemented in the tool TAILOR. However, this work is ongoing.

Flattening Problem Classes While flattening problem instances is a straightforward process, it can raise some difficulties with problem classes. The biggest challenge is the efficient flattening of quantified expressions: since parameters are unspecified, many quantifications cannot be unrolled and quantified subexpression have to be flattened. This can create redundancies, as we will show below.

In our approach, a quantified expressions is flattened to an array of auxiliary variables whose size is derived by the domain of the corresponding quantifiers. For illustration, consider the quantification **forall** $i, j : \text{int}(1..n) . x[i]*x[j] != y[i]*y[j]$ where the quantified subexpression ‘ $x[i]*x[j]$ ’ will be flattened by introducing the array ‘aux’ with n^2 elements (since both i and j range from $1..n$), stated by the constraint ‘ $\text{aux}[(i-1)*(n)+j] \leftrightarrow x[i]*x[j]$ ’ (note that for convenience, we represent auxiliary variable arrays as 1-dimensional arrays). However, in some cases, constraints in quantifications are guarded by a boolean expression, such as $(j!=i)$ in

forall $i, j : \text{int}(1..n) . (j!=i) \Rightarrow (x[i]*x[j] != y[i]*y[j])$

Flattening will introduce n^2 auxiliary variables, however, since $(j!=i)$ will evaluate to *false* in n cases, only $n^2 - n$ auxiliary variables are actually used, the rest is unconstrained. Eliminating this redundancy raises two difficult questions: first, how to generally determine the number of unconstrained auxiliary variables where guard and quantification can be arbitrarily complex? Second, how to best represent an array of auxiliary variables, of which some elements are not used: do we need new data structures or does there exists a general mapping to a simpler data structure? Addressing these questions is an important part of our future work. For now, our investigations have shown that the

introduced redundancy only matters if the auxiliary variables are included into search, otherwise the impact is marginal (for the examples we have considered).

Eliminating Common Subexpressions in Problem Classes Common subexpression elimination as described in previous sections is applicable to both classes and instances. However, since problem classes mainly contain *quantified* common subexpressions, we extend the preprocessing and flattening procedure to deal with quantified expressions.

Preprocessing is extended with *normalising quantifiers*. The aim of normalising quantifiers is to unify quantifiers that range over the same domain which can reveal syntactically common subexpressions: For instance, consider the two quantifications $\forall i \in (1..n) . (x[i] \neq i)$ and $\forall j \in (1..n) . (x[j] \leq y[j])$. Normalisation of quantifiers will re-write j into quantifier i , since they both range over $(1..n)$. However, in the case when two or more quantifiers are used in the same quantification over the same domain, renaming cannot be performed. Therefore we construct a hashmap during preprocessing, that collects all quantifiers that range over the same domain. This hashmap is used during flattening, when matching for syntactically common subexpressions: whenever a quantified subexpression has a no common subexpression, the hashmap is consulted for ‘equal’ quantifiers, and the quantified subexpression is re-written using an ‘equal’ quantifier. As an example, consider the following two quantifications from the basic n -queens model, stating that no two queens may be in the same NW-SE diagonal:

forall $i, j : \text{int}(1..n) \quad (i < j) \Rightarrow (\text{queens}[i] + i \neq \text{queens}[j] + j)$

Both i and j cannot be renamed, so we maintain a hashmap that records that i and j range over the same domain. After flattening ‘queens[i] + i ’ to auxiliary ‘aux0’, the flattening engine will flatten ‘queens[j] + j ’, which will have no common subexpression. When consulted, the hashmap will return i as an ‘equivalent’ quantifier and ‘queens[j] + j ’ will be re-written to ‘queens[i] + i ’. The check for a syntactical common subexpressions will be positive, and ‘queens[j] + j ’ will be represented by ‘aux0’.

7 Common Subexpression Elimination as a Prelude

Common subexpression elimination enables a set of effective local enhancements that provide further model enhancement, as described in this section. Performing these enhancements automatically is ongoing work.

Consequent Decomposition is the decomposition of a complex implication into a conjunction of simpler implications. Specifically, consider the implication $A \Rightarrow \bigwedge_i^{1..n} E_i$ stating that A implies a conjunction of constraints E_i , where A and all E_i are arbitrary relational expressions. It can be decomposed into a conjunction of implications:

$$A \Rightarrow E_1, A \Rightarrow E_2, \dots, A \Rightarrow E_n$$

where subexpression A has several occurrences. This reformulation is beneficial **only** if subexpression A is represented by the same auxiliary variable, i.e. is detected and eliminated by common subexpression elimination. Despite its simplicity, it provides a speedup and can therefore contribute to model enhancement.

Example: Peaceful Army of Queens A peaceful army of queens consists of two equally-sized groups of white and black queens on an $n \times n$ chessboard that do not attack the opposite colour. The objective is to maximise the number of queens on the board. We take the basic model from [22]. Every cell of the board is represented by the 2-dimensional array *board* where each cell can take value 0 (not occupied), 1 (occupied by black) or 2 (occupied by white). The non-attacking constraints of black are

of the form $(cell_{ij}=1) \Rightarrow (\bigwedge_{kl} cell_{kl} < 2)$ stating that if a particular cell is occupied by black, then the cells attacked by that cell ($cell_{kl}$) are not allowed to be white. The non-attacking constraints can be split up into the set of conjunctions

$$(cell_{ij}=1) \Rightarrow (cell_{i_1 j_1} < 2), \dots, (cell_{ij}=1) \Rightarrow (cell_{i_k j_k} < 2)$$

This representation increases the number of constraints, but reduces the number of auxiliary variables, which results in a reasonable speedup (see Section 8).

Tightening Bounds of Auxiliary Variables Common subexpression elimination can trigger tightening of bounds of auxiliary variables. We show this with an example: The Knight’s Tour Problem involves moving a knight on a chess board such that it visits every field exactly once. Such a tour is called *closed* if the knight attacks its initial field in the last move. The model is based on that from Gecode [7] who credit Gert Smolka. The chess board cells are enumerated along the rows; the board width is given by parameter n , the length of the tour by parameter $length$. The knight’s position at each step is represented by the 1-dimensional array ‘tour’ of length $length$, whose domain ranges over the board cells. A knight performs L-shaped moves, i.e. 2 cells in one direction and 1 cell into the orthogonal direction, or vice versa, which is formulated as

$$\text{forall } s : \text{int}(1..length) . (|tour[s]\%n - tour[s + 1]\%n| = 1) \wedge (|tour[s]/n - tour[s + 1]/n| = 2) \\ \vee (|tour[s]\%n - tour[s + 1]\%n| = 2) \wedge (|tour[s]/n - tour[s + 1]/n| = 1)$$

The number of cells moved in horizontal direction is expressed by the difference ‘ $tour[s]\%n - tour[s + 1]\%n$ ’ where ‘ $tour[s]\%n$ ’ represents the column of the knight’s position at step s . We take the absolute value of the difference because both left and right moves are valid. Similarly, the number of cells moved vertically is represented by ‘ $|tour[s]/n - tour[s + 1]/n|$ ’. Hence, the constraint states that a legal move is either a move 1 cell horizontally and 2 cells vertically *or* 2 cells horizontally and 1 cell vertically. The two main common subexpressions in the model are ‘ $|tour[s]\%n - tour[s + 1]\%n| \leftrightarrow aux_1[s]$ ’ and ‘ $|tour[s]/n - tour[s + 1]/n| \leftrightarrow aux_2[s]$ ’. Common subexpression elimination will represent each difference expression with the same auxiliary variable $aux_1[s]$ and $aux_2[s]$ respectively, each ranging from 0 to n . Hence after common subexpression elimination, the constraint will be:

$$\text{forall } s : \text{int}(1..length) . (aux_1[s] = 1) \wedge (aux_2[s] = 2) \vee (aux_1[s] = 2) \wedge (aux_2[s] = 1)$$

This representation highlights two important properties of the auxiliary variables. First, we can derive that both aux_{s1} and aux_{s2} will either be assigned 1 or 2, so we can restrict their bounds to (1..2). Second, the disjunction above represents the fact that the sum of aux_i and aux_j will always be 3, hence we can represent the complex disjunction by a (cheaper) sum constraint: **forall** $s : \text{int}(1..length) . aux_{s1} + aux_{s2} = 3$. The effect of these reformulations is presented in the Experimental Section.

8 Experimental Results

In this section we summarise our empirical analysis of the enhancement techniques during tailoring. First, we investigate enhancements on instance level, then on problem class level. In both cases, we evaluate three tailoring strategies:

- (a) **no enhancement**: standard tailoring without automated enhancement
- (b) **cheap enhancement**: syntactic common subexpression elimination and preprocessing (i.e. restricted semantic equivalence reduction to save time)
- (c) **full enhancement**: applying all proposed enhancement techniques (i.e. investing more time; semantic equivalence detection is reduced to the negation reformulation)

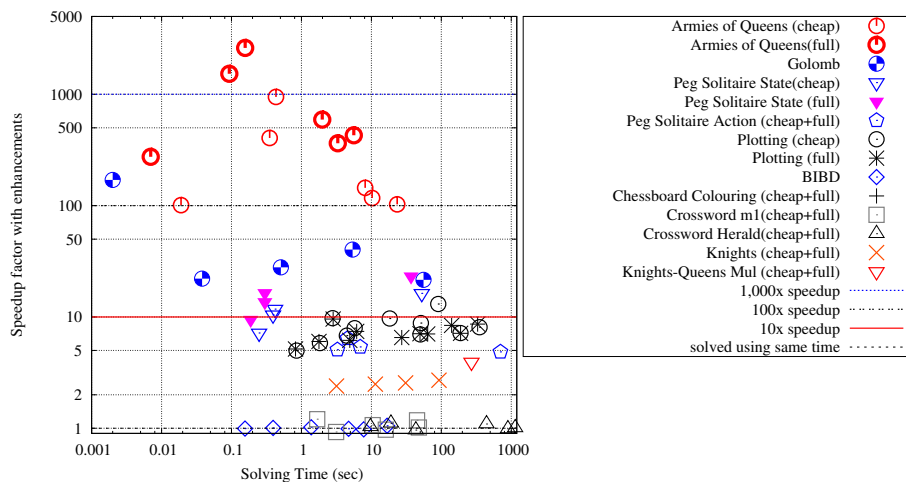


Fig. 2. Speedup in Runtime. The (logarithmic) x -axis represents the solving time *with* enhancement. The y -axis gives the factor to multiply this by to obtain the solving time *without* enhancement, hence instances above $y = 1$ are solved faster with elimination than without. As an example, Armies of Queens instances with cheap enhancement are solved up to 1000 times faster than instances without enhancement

We apply all techniques to different types of constraint problems (optimisation problems, planning problems, puzzles) to demonstrate the general applicability of our work. We use a set of problems in ESSENCE¹ that are available on TAILOR’s website², and examples from the CSP solver competition [3] in XML XCSP 2.1 format [25]. We tailor the instances to MINION input using TAILORv0.3.2 that performs all enhancement techniques automatically (and optionally). For each problem instance, we generate three MINION input files, each tailored in a different way: without enhancement, applying cheap enhancement and full enhancement, using TAILORv0.3.2 with Java REV1.6.0. All instances are solved on the same machine (Dual-core Intel P4 at 3GHz with 1.5Gb RAM) using MINION v0.8.0 using the same variable ordering heuristic (decision variables first, then auxiliary variables) and same value ordering heuristic (ascending).

(1) Experimental Results on Instance Level We are interested in three different features: solving performance, instance size and tailoring time.

Speedups in solving time are summarised in Fig. 2. Cheap enhancement can speedup solving time by up to a *magnitude*, full enhancement even up to 3,500 times. We also observe a vast reduction in search space for some problem families (Peg Solitaire, Armies of Queens, Golomb and Plotting). However, for some instances, such as crosswords, we do not observe a vast reduction in solving time (despite the reduction in auxiliary variables), which illustrates the importance of short tailoring time.

The reduction of auxiliary variables through our enhancements is given in Fig.3. In most cases, we reduce the number of auxiliary variables to 90-40% of the number of auxiliary variables without enhancement. Furthermore, in some problem families, such

² Tailor’s website: <http://www.cs.st-and.ac.uk/~andrea/tailor>

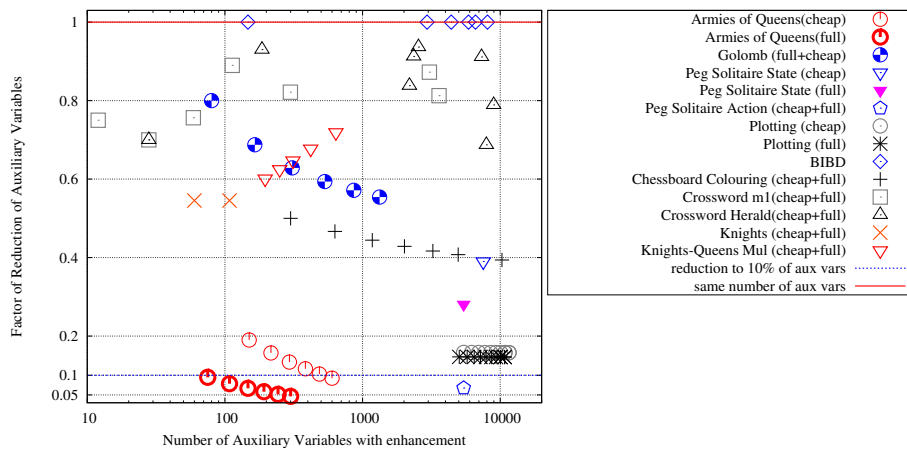


Fig. 3. Reduction of Auxiliary Variables. The x -axis represents the number of auxiliary variables *with* enhancement, and the y -axis the factor of reduction over not enhancing, hence all points below $y=1$ use less variables when applying enhancement techniques. As an example, Armies of Queens instances are below $y=0.2$, hence they have less than 20% auxiliary variables than unenhanced instances. Peg Solitaire instances have the same number of auxiliary variables.

as Plotting or Armies of Queens, cheap enhancement reduces the number of auxiliary variables to 10% - full enhancement even achieves a reduction down to 5%.

Tailoring Time, the most critical feature, is depicted Fig. 4 for both cheap and full enhancement. In general, tailoring times vary between some milli-seconds and a minute and we investigate the differences between tailoring time *with* and *without* performing cheap/full enhancement. With cheap enhancement, we observe a marginal difference of 10% for many instances (in Fig. 4(top), most instances lie between 0.9 and 1.1), a difference that could stem from different runs. We also observe that some problem families are tailored quicker with enhancement than without, such as Plotting instances. For full enhancement, the difference still lies at 10% for many instances. However, tailoring time for some families, such as Armies of Queens, increases with complexity, since additional enhancements can be applied. Therefore, we do not suffer from increased tailoring time, since the investment of some milliseconds/seconds into enhancement is paid off with a reduction of several seconds/minutes when solving the instance.

(2) Experimental Results on Problem Class Level For reasons of space, we restrict our experiments to the problem families of Golomb Ruler, BIBD, Peg Solitaire State and Action. The setup differs from our instance experiments: first, we flatten each problem file to a flat representation (in ESSENCE') (without/with cheap/full enhancement), generating three flat problem class models. Then we tailor them again, together with parameter files and solve the resulting solver instances in MINION.

When investigating problem class enhancement, we are interested in the difference to instance-wise enhancement: we compare tailoring time (see Fig. 5), instance size and solving time (the latter two are not included for reasons of space). We observe that for most problem families, tailoring takes less time on problem class level, with the

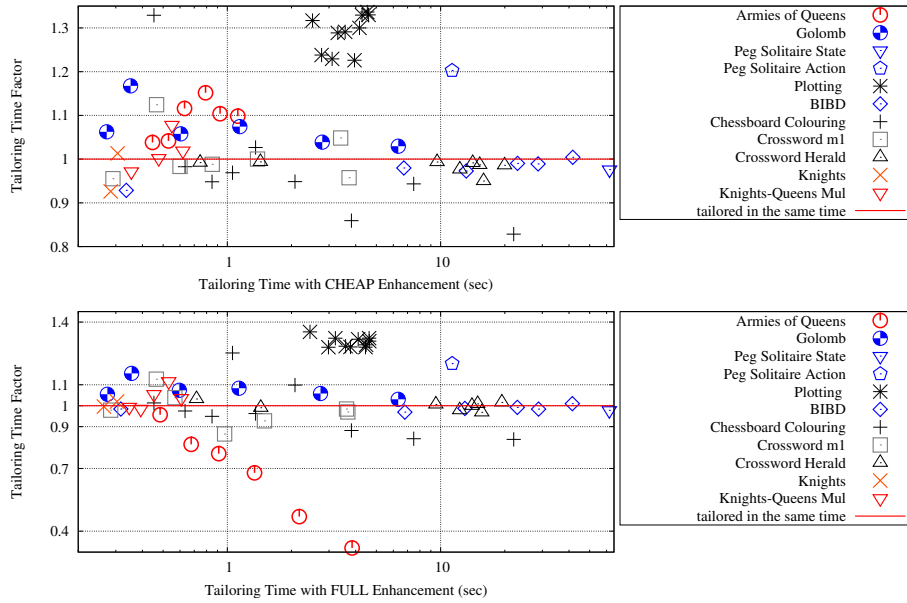


Fig. 4. Tailoring Times. (top) cheap enhancement, (bottom) full enhancement. The x -axis represents tailoring time with enhancement, the y -axis shows the tailoring time factor with enhancement, hence points above $y=1$ depict the cases when tailoring time was reduced when applying enhancements; points below represent cases where tailoring time was increased. As an example, in (top), Plotting instances cover $y=1.3$, hence cheap enhancement reduced tailoring time by 30%.

exception of Golomb Ruler. Most interestingly, tailoring Peg Solitaire Action without enhancement is far more expensive on problem class level than on instance level.

Problem class Tailoring results in almost identical instances as on instance level for BIBD, Peg Solitaire State and Action. Golomb Ruler, however, differs: on one hand, it contains redundancies as described in Section 6, on the other hand, problem class flattening automatically eliminates a particular kind of common subexpressions, even when enhancement is turned off. Solving times are approximately the same (variance of 10%), with the exception of Golomb Ruler, where class-tailored instances are solved quicker (which we cannot explain). In summary, the experiments depict the great potential of problem class enhancement, but shows also, that there are still open questions to investigate.

(3) Discussion of Selected Problem Examples

We use the ‘basic model’ of **Peaceful Army of Queens** from [22] without symmetry breaking constraints in ESSENCE’. Cheap enhancement reduces the instance to about a fifth of its size in average, while also reducing the tailoring time as the problem size increases. The most impressive improvement is given in the number of search nodes and solving time, both reduced by a *magnitude*. Full enhancement achieves even further improvements: we apply consequent decomposition (Section 7) and detect semantic equivalences in form of negations. As expected, tailoring time increases since we invest

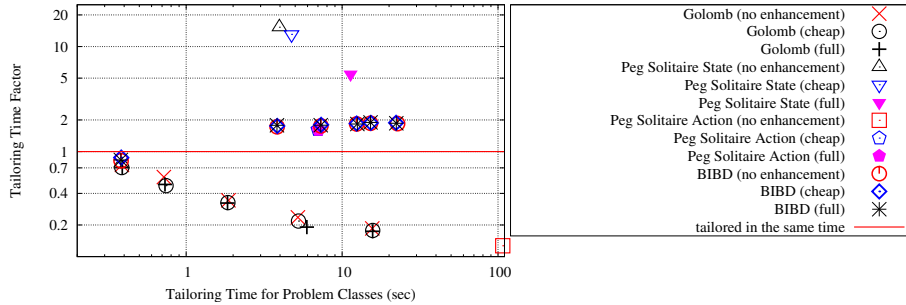


Fig. 5. Tailoring Times on PROBLEM CLASS LEVEL. The x -axis represents tailoring time on problem class level, the y -axis shows the tailoring time factor on class level, hence points above $y=1$ depict the cases when tailoring was cheaper on problem level; points below represent cases where tailoring was cheaper on instance level.

more time into detecting semantic equivalences. However, this investment pays off: full enhancement reduces the number of auxiliary variables to 5% of the original amount, while the number of constraints stays about the same (slightly reduced). The consequences are dramatic: compared to instances without enhancement we gain a speedup in solving time by about 3,500 times

Comparison with results of [22] highlights an important aspect of our work: our results without enhancements are much worse than reported there, while results with it are similar. This is due to the lack of expressiveness in MINION, which, unlike most solvers, does not provide a propagator for linear disequality (hence linear disequality is flattened to binary disequality constraints). However, this only demonstrates, that despite the limitations of the target solver, automated modelling succeeds in generating a most effective instance that is competitive with expert instances from the literature.

We take the basic model of **Golomb Ruler** from [21] that uses quarternary constraints to express the distances between the ticks (see Example in Section 4). Cheap enhancement yields the enhanced distance model from the same paper. This demonstrates how weak models can automatically be enhanced to advanced, effective models from the literature. We do not detect any semantic equivalences during flattening. In average, we gain a $40\times$ speedup in solving time and reduce the search space by 25%.

We use the standard model of **Balanced Incomplete Block Design (BIBD)** (problem 28 in CSPLib [11]) that does not contain common subexpressions, nor any other scope for enhancement. As shown in Fig.4, we do not suffer significantly from attempting cheap or enhancement: tailoring times are approximately the same. This demonstrates the efficiency of our enhancement techniques. For both approaches we generate identical instances and get identical results in terms of time and nodes searched. Fluctuations in search time are presumably just the difference between separate runs. Thus we draw the conclusion that the attempt to eliminate common subexpressions - even in vain - need not significantly slow down the modelling and solving process.

9 Conclusions

We have shown how the tailoring process can be augmented so as to cheaply perform automated enhancement techniques, generating effective constraint instances and

classes. First, we described a general tailoring approach that reduces the overhead introduced by inappropriate flattening. Second, we presented the integration of common subexpression elimination into the flattening procedure, which provides cheap detection and elimination of syntactically common subexpressions. Third, we showed how to detect and reduce simple semantic equivalences into syntactic equivalences, providing further enhancement. Fourth, we extended and adapted these techniques to problem classes and finally, we identified common subexpression elimination as prelude to further enhancement. Our experimental results confirm that all enhancement techniques are cheaply performed (mostly having no significant effect on tailoring time) and can dramatically reduce search space, solving and instance size, for both problem instances and classes.

We stress that all these steps, although powerful and efficient, are *not* routinely performed by constraint solvers or flattening tools at present. Almost all constraint systems perform some translation of the expressions they allow the user to input to match the constraints provided in the system. Thus, the benefits of enhancing techniques during tailoring could be made available in most constraint systems.

References

1. I. Araya, B. Neveau, G. Trombettoni. Exploiting Common Subexpressions in Numerical CSPs. *in CP*,342-357, 2008.
2. Choco: a java library for constraint programming and explanation-based constraint solving <http://choco.emn.fr/>
3. Third International CSP Solver Competition <http://cpai.ucc.ie/>
4. J. Cocks, Global common subexpression elimination, *SIGPLAN*,5:20–24,1970.
5. The ECLiPSe Constraint Programming System <http://eclipse.crosscoreop.com/>
6. C. Fraser, David Hanson A retargetable compiler for ANSI C in *SIGPLAN Notices*,Vol.26(10), pp 29–43, 1991.
7. Gecode: a Generic Constraint Development Environment <http://www.gecode.org>
8. I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. *ECAI*, pp 98–102, 2006.
9. 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.
10. I.P. Gent, I. Miguel, A. Rendl Common Subexpression Elimination in Automated Constraint Modelling. Proceedings of the Workshop on Modeling and Solving, 2008
11. I. P. Gent, T. Walsh. CSPLib: A benchmark library for constraints. Technical Report APES-09-1999, 1999.
12. W. Harvey, P. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints*, 7, pp172–203, 2003.
13. P. Van Hentenryck, L. Michel, L. Perron. Constraint programming in OPL in *PPDP*, 1999
14. T. Latvala, A. Biere, K. Heljanko and T. A. Junttila Simple Bounded LTL Model Checking. *FMCAD*,pp 186-200,2004.
15. T. Le Provost, M. Wallace Generalized Constraint Propagation over the CLP Scheme. *J. Logic Progr.*, 16(3), 1993.
16. D. Marinov, S. Khurshid, S. Bugrara, L. Zhang and M. C. Rinard Optimizations for Compiling Declarative Models into Boolean Formulas in *SAT*, pp 187-202, 2005.
17. B.A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n Queens. *IEEE Expert* 5:16-23, 1990
18. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *CP*, LNCS 4741, 529-543, 2007.
19. D.A. Plaisted, and S. Greenbaum A structure-preserving clause form translation, *Jnl of Computation* 2,293-304
20. H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization *Journal of Global Optimization* 33/4 (2005), 541-562
21. 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.
22. 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.
23. R. Becket, S. Brand, M. Brown, G. J. Duck, T. Feydy, J. Fischer, J. Huang, K. Marriott, N. Nethercote, J. Puchinger, R. Rafeh, P. J. Stuckey, and M. G. Wallace. The Many Roads Leading to Rome: Solving Zinc Models by Various Solvers *ModRef'08*, 2008
24. 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
25. Organising committee of the Third International Competition. of CSP solvers. XML representation of Constraint Networks XCSP 2.1 Format.
26. Benchmarks of the CSP Solver Competition <http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html>