Automatically Enhancing Constraint Model Instances during Tailoring

Andrea Rendl, Ian Miguel, Ian P. Gent
School of Computer Science, University of St Andrews, UK
{andrea, ipg, ianm}@cs.st-andrews.ac.uk

Chris Jefferson
University of Oxford, Oxford, UK
chris.jefferson@comlab.ox.ac.uk

Abstract

Tailoring solver-independent constraint instances to target solvers is an important component of automated constraint modelling. We augment the tailoring process by a set of enhancement techniques of which many are successfully established in related fields, such as common subexpression elimination. Our aim is to apply these techniques in an efficient fashion, since we tailor instance-wise, and not whole problem classes. We integrate automated enhancement into the tailoring procedure, which creates a novel setup with great potential, as our empirical analysis confirms: impressive speedups, additional propagation and instance reduction, all for investing little computational effort.

1. Introduction

Modelling is widely acknowledged as a major bottleneck in the process of solving a problem of interest using a constraint-based approach. 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* constraint instances, a process where a solver-independent constraint model is adapted to a target solver, an important component of our automated modelling approach. 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. Since we consider tailoring problem *instances*, little time should be invested into automated enhancement. Therefore, this paper discusses how enhancement techniques can be integrated into the tailoring process while using little computational effort to enhance constraint instances.

First, we discuss using solver profiles to guide the flattening process so as to avert unnecessary overhead of auxiliary variables or constraints. Second, we present the integration of common subexpression elimination into the flattening procedure, providing cheap detection and elimination of syntactically common subexpressions. Third, we show how to detect and eliminate some semantically common subexpressions, a technique that involves slightly more computational effort, but that can provide further enhancement.

In summary, we present well-known techniques integrated in the tailoring process, resulting in a novel setup. Our experimental results confirm that automated model enhancement succeeds in cheaply producing effective constraint instances from weak models while employing little computational time.

2. Background

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 a set of *constraints* on those variables that a solution must satisfy. A decision variable represents a choice that must be made to solve the problem. The *domain* of potential values associated with each decision variable corresponds to the options for that choice. The second phase consists of using a constraint solver to search for solutions: assignments of values to decision variables satisfying all constraints.

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. Typically, a problem class formulation is paired with a parameter file to create an instance. There exist several successful solver-independent constraint modelling languages such as OPL (Hentenryck, Michel & Perron 1999), MINIZINC (Nethercote *et al* 2007) or ESSENCE' (Gent, Miguel & Rendl 2007). In this paper, all constraint expressions are formulated in ESSENCE'. However, we stress that in our work the choice of modelling language is unimportant.

To solve a solver-independent constraint model, it has to be *tailored* (Gent, Miguel & Rendl 2007) to be suitable for input to a target constraint 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 solvers repertory, which includes flattening of constraints, propagator selection and adaption of heuristics. Most solver-independent modelling languages require some form of tailoring.

A key part of the tailoring process is *flattening*: decomposing a complex constraint expression into an equivalent

conjunction of simpler expressions. The general approach is to replace a subexpression by an auxiliary variable that represents the subexpression, e.g. decomposing $a * b + c{=}0$ into the two constraints $aux{=}a * b$ and $aux + c{=}0$. This is necessary whenever the target constraint solver does not support the original complex constraint expression. Most target solvers require flattened input.

3. General, Efficient Tailoring

A wide range of constraint solvers exist, each with its own strengths and weaknesses, each tackling problem instances in its own way. Problems, where some solvers struggle, others can easily solve and vice versa (CSP Solver Competition 2008). Therefore, automated modelling should not target only one solver, but as many solvers as possible. This can be achieved by generalising the tailoring process.

A main challenge of generalising 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 Prolog (Eclipse 2009), Gecode (Gecode 2009) and MINION (Gent, Jefferson & Miguel 2006). The appropriate constraint representation for each solver is given in the table below:

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

Prolog takes arbitrarily complex expressions, hence no flattening is required. Gecode provides a linear disequality constraint, allowing variables as arguments only, hence we flatten $e * f$ by introducing auxiliary variable aux_1, and post $a + b + c \neq aux_1$. MINION only supports 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. In order to do this, the flattening engine requires information about the solver's constraint repertory - information a *solver profile* can provide.

Solver Profiles

We propose the notion of a *solver profile*, similar to rule-based systems in retargetable compilers (Fraser & Hanson 1991), that captures important features of a particular solver. Those features include variable information (variable and domain types, available data structures), propagator information (constraint type, consistency level, arity, reifiability, etc.), provided 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 the feature is supported or not. For instance, if solver S provides

one n-ary conjunction propagator that is not reifiable, then the the feature *n-ary conjunction* will be set to true, but feature *reifiable n-ary conjunction* will be set to false. Solver profiles can also include solver-specific features, e.g. variable labelling.

Solver Profile-driven Tailoring

Solver profiles can be used to customise tailoring. First, we focus on customising flattening, i.e. using the solver profile's propagator and variable information to direct the flattening procedure. The flattening engine works recursively, i.e. when given an expression, the flattening procedure is again invoked on the expression's arguments. A solver profile can guide the flattening engine: when given an expression, e.g. an n-ary multiplication, 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 only flattened, 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 seach heuristic (in case favoured propagators or search heuristics are not already defined in the modelling language, as possible in MINIZINC). We have extended the tool TAILOR (Gent, Miguel & Rendl 2007) to apply solver profiles during tailoring, currently targeting solvers MINION and Gecode.

Note, that solver profiles cannot provide alternatives in case a constraint is not supported. This can be resolved by either extending the tailoring engine with additional reasoning or extend the modelling language to support the definition of alternative representations (as in MINIZINC).

4. Eliminating Common Subexpressions

In this section we discuss common subexpression elimination, an optimisation technique originating from compiler optimisation (Cocke 1970), that has proven to be powerful in several related disciplines, such as Satisfiability (Marinov *et al* 2005), Model Checking (Latvala *et al* 2004), Proof Theory (Plaisted & Greenbaum 1986) and Numerical CSPs (Araya, Neveu & Trombettoni 2008). We show how to exploit the tailoring process to integrate common subexpression elimination in a computationally cheap way.

Two expressions are called *common* (or equivalent) if they take the same value under all possible satisfying assignments. There exist 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, e.g. expressions $a*b$ and $b*a$ are equivalent. Clearly, syntactically equivalent expressions are also semantically equivalent.

Exploiting explicit linear equalities has been well studied (Harvey & Stuckey 2003; Le Provost & Wallace 1993; Nadel 1990). 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 can be introduced with the intersection of the domains of x and y and 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 derived, either by checking for syntactic or semantic equivalence. The detection of syntactic equivalences is discussed in this section, whereas semantic equivalence is covered in Section 5.

Common Subexpressions in Constraint Instances

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 (Smith, Stergiou & Walsh 1999) is given below:

> **forall** i,j,k,l : int$(1..n)$.
> $((i{<}j) \wedge (k{<}l) \wedge ((j{>}l)\vee(i{>}k))) =>$
> $(ticks[j] - ticks[i] \mathrel{!{=}} ticks[l] - ticks[k])$

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

> $ticks[3] - ticks[1] \mathrel{!{=}} ticks[2] - ticks[1]$
> $ticks[3] - ticks[2] \mathrel{!{=}} ticks[2] - ticks[1]$
> $ticks[3] - ticks[2] \mathrel{!{=}} ticks[3] - ticks[1]$

which contain several occurrences of each subexpression $ticks[2] - ticks[1]$, $ticks[3] - ticks[1]$ and $ticks[3] - ticks[2]$.

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. Furthermore, common subexpression elimination is not routinely done 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 (Choco 2009), do not make use of common subexpressions.

Eliminating Syntactically Common Subexpressions during Flattening

Flattening can be considered a recursive process that, when given an expression, replaces its subexpressions with auxil-

iary variables, if appropriate for the target solver (see Section 3). 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* equivalance). 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: $x - y \le a$ and $x - y \le b$ 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

$$aux = x - y$$
$$aux \le a$$
$$aux \le b$$

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 ($aux \le a$ and $aux \le b$ are still 'only' linear). Linear propagators are very powerful and the number of arguments ($x - y \le a$ or $aux \le a$) 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.

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. Reducing Simple Semantic Equivalence to Syntactic Equivalence

Our approach of eliminating common subexpressions during flattening (see Section 4) is restricted to syntactically common subexpressions. This is because we tailor *instances*, where time cannot be invested into detecting semantically common subexpressions, which is an operation that can be arbitrarily hard. However, typically, there are many simple semantic equivalences that can be easily reduced to syntactic equivalence, and hence detected and eliminated. In this section we discuss two different approaches of detecting and reformulating semantically equivalent subexpressions into syntactically equivalent subexpressions in order to increase the overall number of eliminated common subexpressions.

Detecting and Reducing Semantic Equivalences

We restrict our investigations to a clear-cut set of simple *equivalence relations* (stemming from general reformulation rules, such as commutativity, associativity, logic etc.), that state when two expressions are semantically common. When detecting two semantically equivalent subexpressions, we first identify the preferable representation and then re-write the other subexpression into that representation. For instance, consider detecting common subexpressions $a*b$ and $b*a$: first we pick $a*b$ as preferable representation and re-write $b*a$ into $a*b$, thus generating two syntactically common subexpressions.

However, it is not always clear, which representation is preferable. 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 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. Below we give an overview of the basic reformulations rules and the respective operators:

Reformulation Rule	Detected during
Commutativity $(+,*,\wedge,\vee)$	preprocessing (1)
Associativity $(*)$	preprocessing (1)
Distributivity	flattening (2)
Negation	flattening (2)
Horn Clause Reformulation	flattening (2)
De Morgan's Law	flattening (2)

To reduce tailoring time, we integrate the detection of the first kind of equivalences relations(1) into the preprocessing phase of tailoring, which is cheap to perform. The detection of the second kind(2) 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 time into the procedures. We discuss both approaches in more detail below.

Reducing 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.

Reducing Semantic Equivalences during Flattening

Detecting semantic equivalence can be arbitrarily hard and hence time consuming, especially when dealing with instances only. Therefore we restrict our investigations to simple semantic equivalences. The main idea is to reformulate

expressions with a high potential of semantically equivalent subexpressions 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. 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** The first 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. $\neq, \geq,..$), since they are most likely to contain equivalent subexpressions from our experience. For instance, consider

$$(A = B) \Rightarrow D$$
$$(A \neq B) \Rightarrow C$$

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** Another interesting reformulation to consider is the reformulation from and to Horn Clause representation. 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 $\neg A \lor B$. If neither $A{\Rightarrow}B$, A nor B have a common subexpression, then the alternative representation, $\neg A \lor B$, 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. Similiarily, the law of distributivity can be exploited to create expressions that are likely to match other subexpressions.
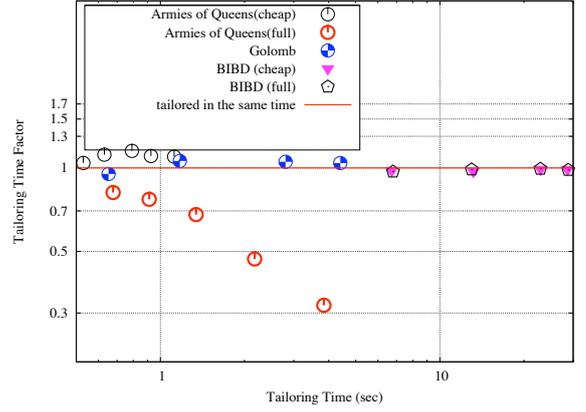

Figure 2: **Tailoring Times**. The x-axis represents the time taken for tailoring with enhancement, the y-axis the tailoring time factor with enhancement. Points above $y = 1$ depict the cases when tailoring time was reduced when applying enhancements; points below represent cases where tailoring time was increased. As an example, tailoring Armies of Queens instances using cheap enhancement takes slightly less time than tailoring without; full enhancement takes longer, increasing with complexity.

6. Local Enhancements

This section covers simple but effective local enhancements, enabled by common subexpression elimination.

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

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

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 reasonable speedup, as illustrated in the Experimental Section (Peaceful Army of Queens).

Enhanced Auxiliary Variable Ordering We can enhance the variable ordering of auxiliary variables by considering those auxiliary variables that represent common subexpressions and put them on top of the variable ordering of auxiliary variables. This is just a slight improvement, since auxiliary variables do not typically play an important role during search.

7. Experimental Results

In this section we summarise the empirical analysis of our enhancement techniques during tailoring. In particular, we evaluate two enhancement strategies:
(a) cheap enhancement: syntactic common subexpression eliminiation and preprocessing (i.e. restricted semantic equivalence reduction to save time)
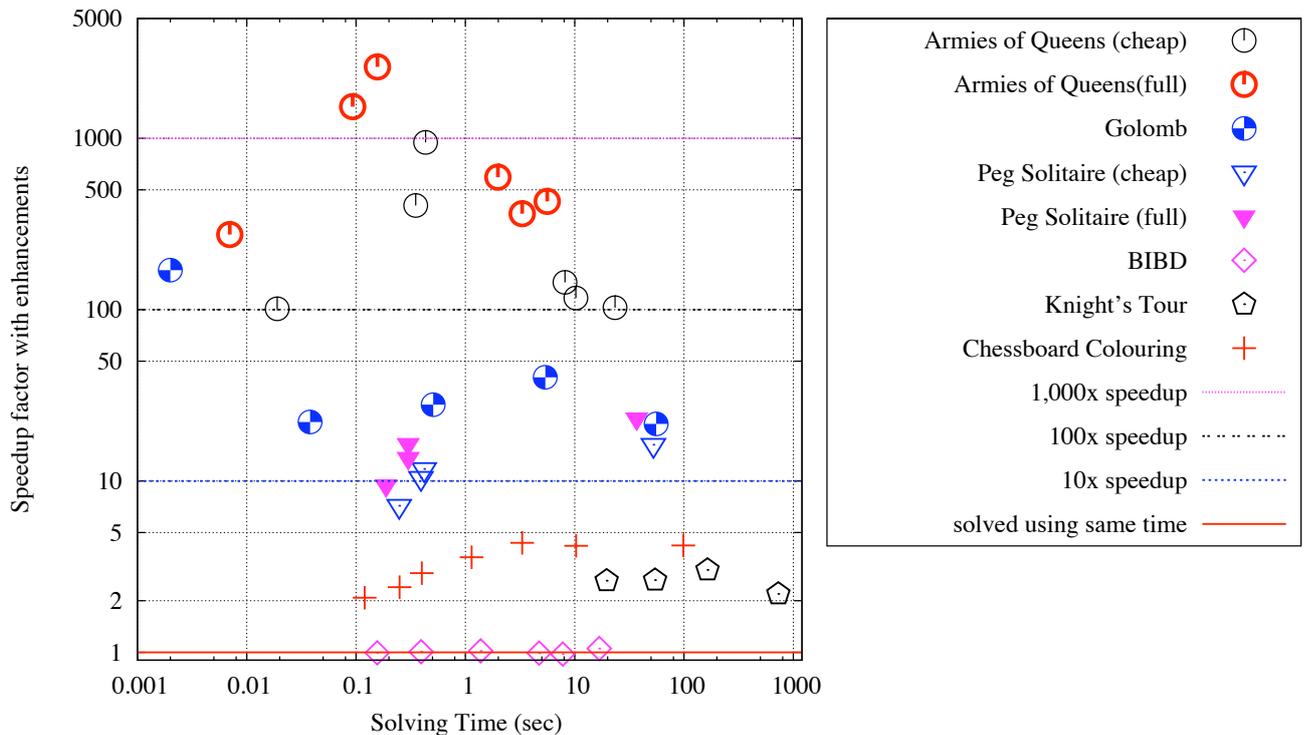**(b) full enhancement**: applying all proposed enhancement

Figure 1: **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. As an example, Armies of Queens instances with cheap enhancement are solved up to 1000 times faster than instances without enhancement; those Armies of Queens instances with full enhancement even up to 2000 times faster. Points above $y = 1$ represent instances which solves faster with elimination than without.

techniques (i.e. investing more time)
We apply both techniques to different types of constraint problems (optimisation problems, planning problems, puzzles) to demonstrate the general applicability of our work. Each problem is formulated in ESSENCE′ without applying symmetry breaking and is available on TAILOR's website[1]. Additionally, we include two examples from the CSP solver competition (CSP Solver Competition 2008) in XML XCSP 2.1 format (XCSP Format 2008). We tailor the instances to MINION input using the tailoring tool TAILORv0.3[2]. TAILOR 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. All instances are solved on the same machine (Dual-core Intel P4 at 3GHz with 1.5Gb RAM) using MINION v0.8.0. We apply the same variable ordering heuristic (decision variables first, then auxiliary variables) and same value ordering heuristic (ascending) in both cases. We are interested in three different features: solving performance, instance size and tailoring time.

Speedups in solving time are summarised in Figure 1.

Cheap enhancement can speedup solving 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 and Golomb). The reduction of auxiliary variables through our enhancements is given in Figure 3: cheap enhancement can reduce the number of auxiliary variables to 10% (Armies of Queens) of the unenhanced instances. Full enhancement even achieves a reduction to 5% (that is 20 times less auxiliary variables). Finally, we investigate the most critical feature: tailoring time (Figure 2). We observe that cheap enhancement does not impair tailoring time that sometimes even decreases by a small factor. Full enhancement however, is more expensive: tailoring Armies of Queens with full enhancement can take up to triple the time of tailoring without. However, the investment of some seconds is worthwile, when saving several minutes during solving.

Golomb Ruler

We model the basic model from (Smith, Stergiou & Walsh 1999) 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
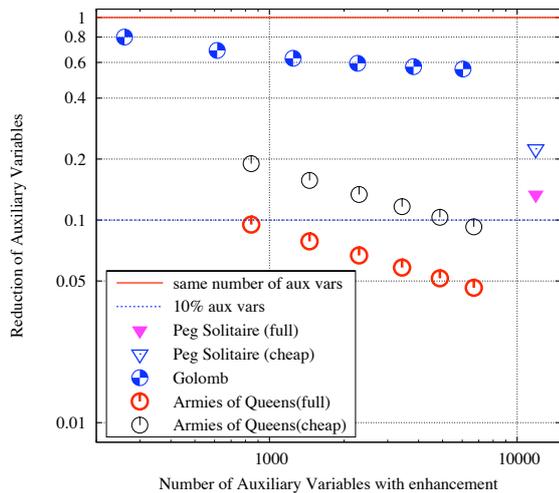
Figure 3: **Reduction of Auxiliary Variables**. The x-axis represents the number of auxiliary variables introduced *with* enhancement, and the y-axis the factor of reduction over not enhancing. As an example, Peg Solitaire instances using cheap enhancement have only 25% auxiliary variables of Peg Solitaire instances tailored without; those using full enhancement just 15%. All points are above $y = 1$, so we always use less variables when applying enhancement techniques. Peg Solitaire instances differ only in the starting hole, so they have the same number of auxiliary variables.

during flattening. In average, we gain a $40\times$ speedup in solving time and reduce the search space by 25%.

Peg Solitaire

We consider the *state*-centric model of Peg Solitaire from (Jefferson, Miguel & Tarim 2006). As depicted in Fig.1, cheap enhancement provides a vast speedup, in average 10 times in solving time. When applying semantically common subexpression elimination during flattening, we detect and eliminate many negated semantical equivalencies. This and the advanced variable ordering result in another speedup of about $2\times$. More details about the sources of common subexpressions in Peg Solitaire can be found in (Rendl, Miguel, Gent & Gregory 2009).

Peaceful Army of Queens

We take the 'basic model' from (Smith, Petrie & Gent 2004) without symmetry breaking constraints in ESSENCE′. First, we investigate syntactic common subexpression elimination that, in average, reduces the instance to about a fifth of its size, 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*.

Secondly, we consider overall enhancement. In this problem model, we apply consequent decompostion (Section 6) and detect semantic equivalences in form of negations. As expected, tailoring time increases (e.g. doubles from 1sec to 2.1sec for n=9) since we invest more time into detect-

ing semantic equivalences. However, this investment pays off: consequent decomposition combined with the elimination of negated common subexpressions 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, compared to syntactic common subexpression elimination $3\times$, using the same search space.

Comparison with results of (Smith, Petrie & Gent 2004) 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 constraints have to be 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.

Balanced Incomplete Block Design (BIBD)

BIBD is problem 28 in CSPLib (Gent & Walsh 1999). We use the standard model from the literature that does not contain common subexpressions, nor any other scope for enhancement. As shown in Fig.2, we do not suffer significantly from attempting cheap or enhancement: tailoring times are approximately the same. The little tailoring time for full enhancement might be suprising. The reason for this lies in the restriction of enhancements: we only attempt to reformulate particularly promising expression types and the BIBD problem model has none. 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. From this experiment we draw the conclusion that the attempt to eliminate common subexpressions - even in vain - need not significantly slow down the modelling and solving process.

Knight's Tour and Chessboard Colouring

We also include benchmarks from the CSP Solver Competition (CSP Solver Competition 2008): the Knight's Tour Problem and the Chessboard Colouring Problem. The instances we use are available at the competition's website (XCSP Instances 2008). Both problems could be solved in half the time using enhancement (results for cheap and full enhancement are the same).

8. Related Work

In their work on interval analysis, Schichl *et al* (Schichl & Neumaier 2005; Vu, Schichl & Sam-Haroud 2004) 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.

Independently of our work[3], Araya *et al* have shown how the idea of common subexpression elimination from compilers can be applied to numerical constraint satisfaction problems (Araya, Neveu & Trombettoni 2008). Their algorithm has similarities to our approach, however it is performed in a different context. Numerical CSPs represent a different set of problems and are solved by different means: numerical variables range over real values instead of discrete values; numerical constraints are arithmetic functions, both linear and nonlinear, such as log or sin, excluding all Boolean relations. Furthermore, numerical CSPs are solved by filtering intervals instead of bounds or domain propagation.

9. Conclusions

We have shown how the tailoring process can be augmented so as to cheaply perform enhancement techniques to generate effective constraint instances. First, we discuss solver profiles, that, when used to guide the flattening process, assist in reducing the overhead (of auxiliary variables or constraints) introduced by inappropriate flattening. Second, we present the integration of common subexpression elimination into the flattening procedure, which provides cheap detection and elimination of syntactically common subexpressions. Third, we show how to detect and reduce semantically common subexpressions into syntactically common subexpressions, a technique that involves slightly more computational effort, but that can provide further enhancement.

Our experimental results show three things. First, both syntactic and semantic common subexpression elimination can have a great effect on both solving time and search space. Second, we can reduce the instance size dramatically which results in a more effective constraint model. Third, we see that eliminating syntactically common subexpressions, even if attempted in vain, does not affect tailoring time. Detecting semantic equivalencies is more expensive, but the additional effort is traded off for additional speedup. For future work, we want to extend our enhancement techniques to whole problem *classes* instead of instances only where more time can be invested into advanced techniques, such as detecting complex semantic equivalences.

Again, 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.

---

[3]Our and their work were first presented at different workshops (Gent, Miguel & Rendl 2008; Araya, Neveu & Trombettoni 2008): the submission date for each was before either event.

References

I. Araya, B. Neveu, G. Trombettoni. Exploiting Common Subexpressions in Numerical CSPs. Small Workshop on Interval Methods, 2008

I. Araya, B. Neveau, G. Trombettoni. Exploiting Common Subexpressions in Numerical CSPS. *in CP*,342-357, 2008.

Choco: a java library for constraint programming and explanation-based constraint solving http://choco.emn.fr/

Third International CSP Solver Competition http://cpai.ucc.ie/

J. Cocke, Global common subexpression elimination, *SIGPLAN*,5:20–24,1970.

The ECLiPSe Constraint Programming System http://eclipse.crosscoreop.com/

C. Fraser, David Hanson A retargetable compiler for ANSI C in *SIGPLAN Notices*,Vol.26(10), pp 29–43, 1991.

Gecode: a Generic Constraint Development Environment http://www.gecode.org

I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. *ECAI*, pp 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*, pp 184-199, 2007.

I.P. Gent, I. Miguel, A. Rendl Common Subexpression Elimination in Automated Constraint Modelling. Proceedings of the Workshop on Modeling and Solving, 2008

I. P. Gent, T. Walsh. CSPLib: A benchmark library for constraints. Technical Report APES-09-1999, 1999.

W. Harvey, P. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints*, 7, pp172–203, 2003.

P. Van Hentenryck, L. Michel, L. Perron. Constraint programming in OPL *in PPDP*, 1999

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.

T. Latvala, A. Biere, K. Heljanko and T. A. Junttila Simple Bounded LTL Model Checking. in *FMCAD*, pp 186-200, 2004.

T. Le Provost, M. Wallace Generalized Constraint Propagation over the CLP Scheme. J. Logic Programming, 16(3), 1993.

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.

B.A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n Queens. *IEEE Expert* 5:16-23, 1990

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.

D.A. Plaisted, and S. Greenbaum A structure-preserving clause form translation, . In *Journal of Computation* 2, 293-304

A. Rendl, I. Miguel, I.P.Gent and P. Gregory Enhancing Constraint Models of Planning Problems by Common Subexpression Elimination *Circa*-preprint 2009-3, 2009

H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization Journal of Global Optimization 33/4 (2005), 541-562

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.

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.

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

Organising committee of the Third International Competition. of CSP solvers. XML representation of Constraint Networks XCSP 2.1 Format.

Benchmarks of the CSP Solver Competition http://www.cril.univ-artois.fr/ lecoutre/research/benchmarks/benchmarks.html