

Constraint Model Enhancement by Automated Common Subexpression Elimination

student: Andrea Rendl
supervisors: Ian Miguel and Ian P. Gent

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

Abstract. The modelling bottleneck in Constraint Modelling prevents the widespread use of Constraint Programming techniques. Automated Constraint Modelling addresses this problem. To enhance automatically generated models, we eliminate common subexpressions during the modelling process, as compilers do when compiling source code. Common subexpression elimination can lead to a dramatic reduction in the size of a constraint problem, as well as a reduction in solving time and the same number of search nodes. Furthermore, it can lead to enhanced propagation and reduced search. Thus we propose common subexpression elimination as an important technique for Constraint Programming.

1 Introduction

One approach to automated constraint modelling is to refine a high-level problem specification to a set of CSPs. The automated refinement system CONJURE [1] takes a problem specification in the problem specification language ESSENCE[2] and refines it to a CSP in the solver-independent modelling language ESSENCE'. The tailoring system TAILOR [3] adapts solver-independent CSPs to a target solver. Hence we have three levels of abstraction: on top the specification level (ESSENCE), the solver-independent level on intermediate level (ESSENCE') and the solver-adapted level (target solver input language). In this paper, we use MINION [4] as target solver.

Our aim in this paper is to enhance problem models during *tailoring*, i.e. at the transition from solver-independent to solver-adapted level. Solvers take either problem instances or problem classes as input. An instance is obtained from a class by giving a value for each parameter in the model (e.g. with $n = 8$, we obtain the 8-queens instance). Our focus lies on enhancing problem instances.

Common subexpression elimination is a successful technique from Compiler Construction [5]: the idea is to omit identical pieces of code to enhance a program. We show how this idea can be applied to Constraint Modelling and how it can significantly improve a model's performance.

2 Normalisation

Normalisation is a preprocessing step to facilitate common subexpression detection (see Section 3). It consists of two basic components: evaluation and ordering

of expressions. Please note that we do not apply any kind of factorisation of expressions.

Ordering We impose an order $<_o$ on expressions in ESSENCE' and transform every model to minimal form with respect to $<_o$. The order reflects the complexity of constraints: constants are on top of the order, followed by variables and arrays. Linear expressions come before non linear expressions. Expressions are ordered starting from the leaves of the expression tree. An equality constraint, for example, is ordered by examining the left argument first, followed by the right. To illustrate, consider the following normalisation:

$$x_1 * x_2 = x_4 + x_3 \longrightarrow x_3 + x_4 = x_1 * x_2$$

Evaluation Evaluation is particularly effective when tailoring instances since all parameters have been instantiated. We start evaluation from the leaves to the expression root. Expressions that are evaluated to *true* are discarded and if an expression is evaluated to *false*, we denote the instances unsatisfiable. Extensive evaluation is costly so we perform cheap evaluation steps only, such as constant evaluation, simple algebraic transformations, basic logic and singleton domain propagation. Singleton domain propagation is performed when a variable's domain ranges only over a single value, so every occurrence of the variable can be replaced by the corresponding value. Another form of simple evaluation is unary constraint propagation (enforcing node consistency). Unary constraints in problem instances only restrict the corresponding variable's domain, so they are easy to evaluate. The examples below illustrate different evaluation types we perform. Please note that x stands for a variable, c for a constant, $ub(x)$ returns x ' upper bound and $lb(x)$ returns the lower bound.

Expression	Result	Restrictions	Description
$3 * 4 - 2$	10		<i>Constant Evaluation</i>
$exp + 0$	exp		<i>Algebraic Identity</i>
$exp - exp$	0		<i>Algebraic Inverse</i>
$exp \wedge false$	<i>false</i>		<i>Logic</i>
x	c	x 's domain is $(c..c)$	<i>Singleton Domain Propagation</i>
$x > c$	<i>false</i>	$ub(x) < c$	<i>Unary Constraint Propagation</i>
	<i>true</i>	$lb(x) > c$	
	$lb(x) = c$	$lb(x) \leq c$	

3 Common Subexpression Elimination

The idea of common subexpression elimination is to find two common (equivalent) expressions and replace every occurrence of the more expensive¹ expression by the other, less expensive expression to enhance the model. It originates from Compiler Construction [5]. We say two expressions are *common* if they take the same value under all possible satisfying assignments. We define *syntactically* and *semantically* equivalent expressions. Syntactically equivalent expressions are

¹ *expensive* in the context of the expression's complexity

written the same way, for instances the two products $a * b$ and $a * b$. Semantically equivalent expressions *mean* the same thing i.e. the equivalence can be deduced by the operational semantics, such as with the equivalence relation $a * b = c$. Due to properties such as commutativity, many semantically equivalent logical and arithmetic expressions can be written so as to be syntactically distinct. A simple example is $a * b$ versus $b * a$. By normalising a constraint model (as we describe in Section 2) prior to common subexpression detection, the test for semantic equivalence is, in many cases, therefore reduced to the much cheaper test for syntactic equivalence.

3.1 Explicitly Common Subexpressions

We call two expressions explicitly common if the equivalence is stated explicitly in the model, for instance with $e_1 = e_2$. The simplest case of explicit common subexpressions $x = y$ is where x and y are atomic expressions, i.e. variables or constant values. The standard enhancement approach is to use x for every occurrence of y and, if y is a variable, to remove y from the set of variables, thus saving a variable. This approach has been extensively studied in [7, 15, 16]. The general case of explicit common subexpressions $X = Y$ occurs when X and Y are arbitrarily complex expressions. As example, consider the expression $x + y = y * z$. According to our ordering, $x + y$ is cheaper than $y * z$ and we can replace every further occurrence of $y * z$ with $x + y$.

3.2 Common Subexpressions During Flattening

Flattening is the process of breaking a complex expression into an equivalent conjunction of simple expressions. It is required if the target solver does not support a particular complex expression. Flattening naturally introduces a large number of equalities and reification constraints, which are a rich source of common subexpressions. If a certain subexpression appears again, we can simply *re-use* the auxiliary variable that already represents the subexpression, as the simple example below demonstrates:

Unflattened	Standard Flattening	Flattening with Common Subexpression Elimination
$a + x * y = 0$ $x * y + b = t$	$aux_1 = x * y$ $a + aux_1 = 0$ $aux_2 = x * y$ $aux_2 + b = t$	$aux_1 = x * y$ $a + aux_1 = 0$ $aux_1 + b = t$

An important source of common subexpressions are quantified expressions that are unrolled during flattening: common subexpressions arise between the expressions gained from different values of the quantification. In our implementation, we flatten expressions bottom-up, i.e. expressions are flattened starting from the leaves of the expression tree. We maintain a hashmap that maps all previously flattened subexpressions to their corresponding auxiliary variables. Whenever we flatten a new subexpression we look up the hashmap for an equivalent expression: if we find an equivalent expression, we replace it with the respective auxiliary

r, c, n	Tailoring (s)		Solving (s)		Search Nodes	
	♠	♡	♠	♡	♠	♡
5-5-2	0.39	0.66	0.25	0.12	17,367	17,367
6-6-2	0.53	0.53	0.60	0.25	28119	28119
7-7-2	0.77	0.72	1.16	0.40	39,807	39,807
8-8-2	0.89	0.86	4.09	1.14	98,367	98,367
9-9-2	1.14	1.08	14.41	3.31	239,231	239,231
10-10-2	1.51	1.52	43.13	10.31	573,951	573,951
12-12-2	2.52	2.36	414.44	98.63	3,193,855	3,193,855

r, c, n	Aux Variables		Constraints		Common Sub-expressions
	♠	♡	♠	♡	
5-5-2	600	300	700	400	300
6-6-2	1,350	630	1,575	855	720
7-7-2	2,646	1,176	3,087	1,617	1,470
8-8-2	4,704	2,016	5,488	2,800	2,688
9-9-2	7,776	3,240	9,072	4,536	4,536
10-10-2	12,150	4,950	14,175	6,975	7,200
12-12-2	26,136	10,296	30,492	14,652	15,840
20-20-3	216,600	79,800	252,700	115,900	136,800

Table 1. Chessboard Colouration with (♡) and without (♠) common subexpression elimination

variable. This approach reduces the time required to match subexpressions and the memory we spend to collect previously flattened subexpressions.

The benefits we gain are great. First, if an instance contains common subexpressions of this kind, we save a variable and a set of constraints (depending on the complexity of the common subexpression) for every subexpression. We obtain a second large benefit, with even greater potential. This is that we can get additional propagation through re-using auxiliary variables. Furthermore, if an instance has no common subexpressions, we do not lose significant time in the attempt to eliminate common subexpressions [6]. Experimental results in Section 4 demonstrate the power of common subexpression elimination.

Some solvers flatten their input themselves, such as the Eclipse Constraint Programming System [8] which does not eliminate common subexpressions [9]. However, most solvers, such as MINION or Gecode [10] take a flattened model as input, hence flattening (in combination with common subexpression elimination) has to be done by the modeller - a tedious task, even for an expert. Therefore both Constraint novices and experts benefit from automated common subexpression elimination: poor models are drastically improved and good models might be improved if they contain common subexpressions without increasing tailoring time significantly.

4 Experimental Results

In this section we compare instances generated with and instances generated without common subexpression elimination. All instances are formulated in XCSP format [11] and come from the benchmark repository of the XCSP solver competition [12]. To tailor the instances to MINION we use the tool TAILOR in which we have implemented (optional) common subexpression elimination. For each problem instance, we generate two different MINION input files: one that is

tailored by eliminating common subexpressions and one that is not. Both models are solved on the same machine (Dual-core Intel P4 at 3GHz with 1.5Gb RAM) using MINION v0.6. We apply the same variable ordering heuristic (decision variables first, then auxiliary variables) and the same value ordering heuristic (ascending) in both cases and search for all solutions. We compare solving performance as well as the tailoring time and give an overview about the number of constraints and auxiliary variables in each instance. More experimental results (that demonstrate a speedup of up to a magnitude) can be found in [6].

Knight’s k -Tour The Knight’s k -Tour problem is to move a knight (according to the rules of chess) on an empty $n \times n$ chessboard such that the knight reaches its initial position after k distinct moves. Table 2 shows the results: instances with eliminated common subexpressions are solved about 3 times faster on a slightly reduced search space. All instances have 30 common subexpressions and tailoring time was the same for both types of instances.

Chessboard Colouring Problem The Chessboard Colouration problem is to fill a $r \times c$ chessboard with n colours such that the corner fields from every rectangle on the chessboard have to be assigned to different colours. Instances tailored with common subexpression elimination are solved about 4 times faster than instances without common subexpression elimination (see Table 1) on the same search space. Table 1 also illustrates the amount of auxiliary variables and constraints as well as the amount of common subexpressions detected in each instance.

5 Related Work

The Zinc [13]/Cadmium [14] system has a similar approach to ESSENCE / CONJURE, but with the focus on merging different techniques with Constraint Programming, such as Integer Programming. Le Provost and Wallace discuss derivation and elimination of common subexpressions during propagation in [15] but restrict their discussion to explicit atomic subexpressions. Harvey and Stuckey present eliminating explicit atomic and linear subexpressions in their work on improving linear constraint representations in [16]. Neither addresses common subexpression elimination during flattening nor elimination of non-linear constraints, as we do in our work. In their work on interval analysis, Schichl *et al* [17, 18] discuss common subexpression elimination in models of mathematical problems represented as directed acyclic graphs. These studies have much

n,k	Solving Time (s)		Search Nodes	
	♠	♡	♠	♡
10,5	51.38	19.63	1,964,189	1,963,504
12,5	143.48	54.39	4,814,685	4,813,774
15,5	497.02	164.08	13,721,664	13,720,414
20,5	>800.00	729.23	>19,776,491	50,074,974

Table 2. Knight’s Tour instances with(♡) and without(♠) common subexpression elimination

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.

6 Conclusion and Future Work

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

For future work we intend to classify sources of common subexpressions and apply common subexpression elimination on problem classes and during refinement of abstract specifications in CONJURE.

Acknowledgements Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. Andrea Rendl is supported by a DOC fFORTE scholarship from the Austrian Academy of Sciences and UK EPSRC grant EP/D030145/1.

References

1. A. Frisch, C. Jefferson, B. Martínez Hernández, and I. Miguel. The rules of constraint modelling. *IJCAI*, pp 109–116, 2005.
2. A.M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of Essence: A constraint language for specifying combinatorial problems. *IJCAI*, pp80-87, 2007.
3. 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.
4. I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. *ECAI*, pp 98–102, 2006.
5. J. Cocke, Global common subexpression elimination, *SIGPLAN Not.*, 5:20–24, 1970.
6. I.P. Gent, I. Miguel, A. Rendl Common Subexpression Elimination in Automated Constraint Modelling *Proceedings of the Workshop on Modeling and Solving Problems with Constraints, 2008, to appear*
7. B.A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n Queens. *IEEE Expert* 5:16-23, 1990
8. The ECLiPSe Constraint Programming System <http://eclipse.crosscoreop.com/>
9. Warwick Harvey Personal Communication, May 2008
10. Gecode: a Generic Constraint Development Environment <http://www.gecode.org>
11. XML format XCSP 2.1, <http://www.cril.univ-artois.fr/CPAI08/XCSP2.1.pdf>
12. Third International CSP Solver Competition, (CSP, Max-CSP and Weighted-CSP competition), 2008 <http://www.cril.univ-artois.fr/CPAI08>
13. M. de la Banda, K. Marriott, R. Rafeh, M. Wallace. The modelling language Zinc. *CP*, 700-705, 2006.
14. P. J. Stuckey, M. de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, T. Walsh. The G12 Project: Mapping Solver Independent Models to Efficient Solutions. *CP*, 13-16, 2005.
15. T. Le Provost, M. Wallace Generalized Constraint Propagation over the CLP Scheme. *J. Logic Programming*, 16(3), 1993.
16. W. Harvey, P. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation, *Constraints*, 7, pp172–203, 2003.
17. H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization *Journal of Global Optimization* 33/4 (2005), 541-562
18. 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

This article was processed using the L^AT_EX macro package with LLNCS style