Artificial Intelligence 251 (2017) 35–61

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Automatically improving constraint models in Savile Row

Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, Patrick Spracklen

School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK

ARTICLE INFO

Article history:
Received 10 February 2016
Received in revised form 16 June 2017
Accepted 5 July 2017
Available online 13 July 2017

Keywords:
Constraint satisfaction
Common subexpression elimination
Modelling
Reformulation
Propositional satisfiability

ABSTRACT

When solving a combinatorial problem using Constraint Programming (CP) or Satisfiability (SAT), modelling and formulation are vital and difficult tasks. Even an expert human may explore many alternatives in modelling a single problem. We make a number of contributions in the automated modelling and reformulation of constraint models. We study a range of automated reformulation techniques, finding combinations of techniques which perform particularly well together. We introduce and describe in detail a new algorithm, X-CSE, to perform Associative–Commutative Common Subexpression Elimination (AC-CSE) in constraint problems, significantly improving existing CSE techniques for associative and commutative operators such as $+$. We demonstrate that these reformulation techniques can be integrated in a single automated constraint modelling tool, called Savile Row, whose architecture we describe. We use Savile Row as an experimental testbed to evaluate each reformulation on a set of 50 problem classes, with 596 instances in total. Our recommended reformulations are well worthwhile even including overheads, especially on harder instances where solver time dominates. With a SAT solver we observed a geometric mean of 2.15 times speedup compared to a straightforward tailored model without recommended reformulations. Using a CP solver, we obtained a geometric mean of 5.96 times speedup for instances taking over 10 seconds to solve.

1. Introduction

In numerous contexts today we are faced with making decisions of increasing size and complexity, where many different considerations interlock in complex ways. Consider, for example, a staff rostering problem to assign staff to shifts while respecting required shift patterns and staffing levels, physical and staff resources, and staff working preferences. The decision-making process is often further complicated by the need also to optimise an objective, such as to maximise profit or to minimise waste.

It is natural to characterise such problems as a set of decision variables, each representing a choice that must be made in order to solve the problem at hand (e.g. which staff member is on duty for the Friday night shift), and a set of constraints describing allowed combinations of variable assignments (e.g. a staff member cannot be assigned to a day shift immediately following a night shift). A solution is an assignment of a value to each variable satisfying all constraints. Many decision-making and optimisation formalisms take this general form, including: constraint programming (CP) [61], propositional satisfiability (SAT) and its extensions [10], and operations research approaches, particularly Mixed Integer Pro-

E-mail addresses: pwn1@st-andrews.ac.uk (P. Nightingale), ozgur.akgun@st-andrews.ac.uk (Ö. Akgün), ian.gent@st-andrews.ac.uk (I.P. Gent), caj21@st-andrews.ac.uk (C. Jefferson), ijm@st-andrews.ac.uk (I. Miguel), jlps@st-andrews.ac.uk (P. Spracklen).

http://dx.doi.org/10.1016/j.artint.2017.07.001
0004-3702/

gramming (MIP) [32]. These approaches have much in common, but differ in the types of decision variables and constraints they support, and the inference mechanisms used to find solutions.

In all of these formalisms the *model* of the problem is crucial to the efficiency with which it can be solved. A model in this sense is the set of decision variables and constraints chosen to represent a given problem. There are typically many possible models and formulating an effective model is notoriously difficult [54]. Therefore, automating modelling is a key challenge. In this paper we investigate automated reformulation of models when targeting CP and SAT solvers, with the goal of reaching an efficient model by a sequence of reformulations starting with a simple model. We investigate a range of reformulation techniques and some of their interactions, and show in some cases that a very high performance model can be created in this way from a naïve model.

1.1. Constraint programming, propositional satisfiability, and tailoring

CP and SAT both follow the general template described above, in that an assignment is sought for a set of variables to satisfy a set of constraints. The key difference is in how CP and SAT represent a given problem. In SAT variables are uniformly Boolean and the constraints are Boolean formulae, expressed in conjunctive normal form [10]. CP provides a richer language of discrete variables with domains either given in extension or expressed in terms of upper and lower bounds, arithmetic and logical operators over these variables, and a library of 'global' constraints that capture common reasoning patterns.

The two formalisms have complementary strengths. The simplicity of SAT supports the implementation of highly efficient solvers, employing techniques such as conflict-driven learning, watched literals, restarts and dynamic heuristics for backtracking solvers [44], and sophisticated incomplete techniques such as stochastic local search [62]. These solvers are able reliably to find solutions to SAT problems with millions of variables, and as a result SAT has many important applications, such as hardware design and verification, planning, and combinatorial design [40]. The richer language of CP enables concise models, and powerful reasoning driven by global constraints has also given rise to a wide variety of successful applications, including configuration, vehicle routing, scheduling and planning [61].

The importance of modelling is well recognised in both fields. In CP a number of constraint modelling languages have been developed, such as OPL [68], MiniZinc [65] and Essence Prime [49]. These languages provide facilities to model parameterised classes of problems (where an individual problem instance is specified by giving values for the class parameters), such as arrays and iteration or comprehension, and to abstract away from the specific details of the many available individual constraint solvers. When targeting SAT, most effort has focussed on constructing performant encodings (i.e. translations into SAT) of classes of constraints such as the allDifferent or arithmetic constraints [53,69,29,63,67]. There has been little attention on the issue of translating a higher level modelling language effectively to SAT.

Tailoring [60] is the process of taking a constraint model of a problem class in a solver-independent modelling language and a value for each of the class parameters, and translating the problem instance into a form suitable for a given solver. We focus herein on reformulation of the constraint modelling language Essence Prime [49], using the constraint modelling assistant Savile Row [47,50]. Essence Prime is similar to other constraint modelling languages such as OPL and MiniZinc. Savile Row is able to translate Essence Prime to target CP and SAT solvers. Tailoring must be efficient: it is performed separately for each problem instance, hence any computationally expensive reformulation performed during tailoring must pay for itself by saving time during solving. Tailoring presents an opportunity to reformulate a model to improve it in a number of ways, some of which are illustrated in our example below.

1.2. Motivating example

We use Balanced Incomplete Block Design (BIBD) [52] as a motivating example to illustrate four quite different reformulations working together to dramatically improve a simple model. All four reformulations are automated in Savile Row. BIBD is a problem class with three integer parameters v, k and λ, and two constants derived from the parameters $b = \frac{\lambda v(v-1)}{k(k-1)}$ and $r = \frac{\lambda(v-1)}{k-1}$. The problem is to place each of v *objects* into a subset of the b *blocks*, so that each block contains k distinct objects, each object occurs in exactly r blocks, and every two distinct objects occur together in exactly λ blocks. The objects and blocks are both interchangeable. The BIBD problem arose from experimental design theory. We use Meseguer and Torras's first model [43], extended with Lex^2 symmetry breaking constraints [19] (i.e. constraints that forbid a set of solutions that are symmetric to other solutions). The model has a v by b matrix m of Boolean decision variables, where the variable $m[i, j]$ indicates whether object i is contained in block j. The model is described in detail in Section 5.9.1.

Savile Row first substitutes the values of v, k, λ, b and r throughout. Then the matrix m is replaced with vb individual Boolean variables, and all quantifiers and comprehensions are unrolled to construct the concrete set of constraints. The model is now almost suitable for output to a conventional constraint solver. However, there is an opportunity to substantially improve the model before it is specialised for a solver.

Each decision variable has a set of possible values that it may take, called its *domain*. The first reformulation step is to filter the domains of the decision variables, as described in Section 5.4. Domain filtering removes values from the variable domains that cannot take part in any solution. Consider the problem instance where $v = 8, k = 4, \lambda = 6$. Domain filtering assigns 70 of the original 224 decision variables in the matrix m, as shown in Fig. 1. The second step is variable unification, as described in Section 5.4. Variable unification removes decision variables in a number of cases. In this case each of the deleted variables is equal to a constant. For example, $m[1, 1]$ is replaced by 0 throughout. After variable unification many of

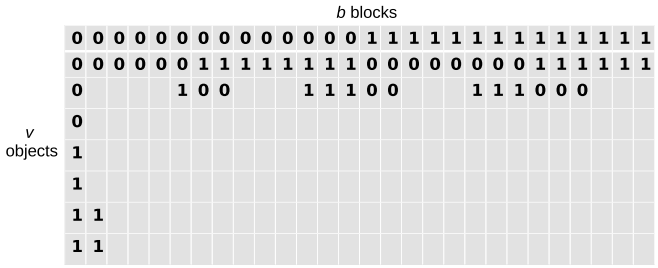

Fig. 1. Variables assigned by domain filtering of BIBD $v = 8, k = 4, l = 6$.

the constraints contain constants. Each expression type in Savile Row has a *simplifier* (described in Section 3.3). Simplifiers will at least evaluate constant expressions, but can in some cases do significantly more than that. In this case, a sum expression (representing a scalar product constraint between the first and fifth rows) may be simplified.

$$0 \times 1 + 0 \times m[5,2] + \cdots + 0 \times m[5,14] + 1 \times m[5,15] + \cdots + 1 \times m[5,28] = 6$$

This expression simplifies to a much shorter expression that contains no products:

$$m[5,15] + \cdots + m[5,28] = 6$$

The first two rows and parts of the third row are assigned, allowing many of the scalar product constraints to be simplified. Each row has a sum constraint such as:

$$m[5,1] + m[5,2] + \cdots + m[5,14] + m[5,15] + \cdots + m[5,28] = 14$$

The fourth step is to extract common subsets of terms between row constraints and (simplified) scalar product constraints. This is a type of common sub-expression elimination (as described in Section 2 below). The common subset $m[5,15] + \cdots + m[5,28]$ is represented by a new variable a, resulting in the following three constraints.

$$\begin{aligned} m[5,1] + m[5,2] + \cdots + m[5,14] + a &= 14 \\ a &= 6 \\ a = m[5,15] + \cdots + m[5,28] & \end{aligned}$$

Variable unification replaces a with 6 throughout, leading to the following:

$$\begin{aligned} m[5,1] + m[5,2] + \cdots + m[5,14] &= 8 \\ m[5,15] + \cdots + m[5,28] &= 6 \end{aligned}$$

Meseguer and Torras's model was improved by Frisch et al. [23] by the addition of a set of *implied constraints* (i.e. redundant constraints that are entailed by the original set of constraints), creating the best published model (to our knowledge). Savile Row automatically generates a superset of the implied constraints proposed by Frisch et al., and consequently the Savile Row model outperforms the Frisch et al. model in our experiments in Section 5.9.1. This example illustrates a number of simple reformulations combining to substantially improve the formulation of a constraint model.

1.3. Contributions

We make three main contributions in this paper.

1. We evaluate a range of reformulations for use in automated modelling. Common Sub-expression Elimination (CSE) is a particular focus of this paper [60,59,2]. CSE is a type of constraint reformulation that has proved to be useful on a range of problems, including both finite-domain and numerical constraint problems. We contribute new algorithms for Identical CSE and Active CSE that improve on existing work. Furthermore, we have discovered sets of reformulations that together are much more effective than any single reformulation.
2. We introduce and describe in detail a new algorithm, X-CSE, to perform AC-CSE (Associative–Commutative Common Sub-expression Elimination). This uses the fact that operators such as + are associative and commutative: expressions containing such operators can be rearranged to reveal additional common subexpressions (CSs). We show that X-CSE is able to find CSs automatically in a variety of problems, and that using these subexpressions can greatly reduce search and improve solving time. A particular advantage of X-CSE is that it is able to find and exploit small CSs that occur in many constraints, as well as larger ones that occur in few constraints. We show that X-CSE can be particularly effective in combination with other reformulations, specifically an automated reformulation of allDifferent constraints leading to sum constraints, and when filtering domains using a variant of Singleton Arc Consistency. When applying X-CSE we demonstrate substantial gains in performance.

3. We demonstrate that the reformulations described above can be integrated in a single automated constraint modelling tool, called Savile Row, whose architecture we describe. We use Savile Row as an experimental testbed to evaluate each reformulation on a set of 50 problem classes with 596 instances in total. We describe a very extensive set of experiments where reformulations are switched on one by one and their effects analysed as part of the system.

2. Background and related work

There are many ways of improving constraint models, some requiring manual interaction [5], and others that are entirely automatic [24]. For example these methods can discover global constraints or automatically detect and remove symmetries [42]. These improvements often complement each other, for example Frisch, Jefferson, and Miguel [23] show how breaking symmetries can lead to effective implied constraints for BIBDs. In this section we review the most relevant related work.

2.1. Variable unification

Unifying equal variables is a standard technique implemented by a number of tools including MiniZinc [65], Savile Row [47,50] and IBM CP Optimizer Presolve [36]. Details of our approach to variable unification are given in Section 5.4.

2.2. Variable domain filtering

Filtering the domains of decision variables is a recognised preprocessing technique in constraint programming. Domain filtering removes values that cannot take part in any solution. Strong consistencies may be used to tighten the formulation of an instance [7,37]. Domains may be filtered by some form of propagation prior to (and potentially enabling) other reformulations. IBM CP Optimizer Presolve uses propagation to obtain tighter bounds for variables and expressions, and redundant constraints are removed before search [36]. The strength of propagation is unspecified.

We have previously [47,50] filtered the user-defined decision variables using a strong consistency related to Singleton Arc Consistency (SAC) [7]: SACBounds applies the SAC test to prune the upper and lower bound of each variable to exhaustion. The Minion solver [30] was used to establish SACBounds. Leo and Tack filtered domains of both user-defined and introduced decision variables in MiniZinc [39], using the standard level of propagation of the solver Gecode [28]. Variable domain filtering has proven to be valuable alone [39] and as an enabler of other reformulations [47,50]. Our approach to domain filtering is described in Section 5.4.

2.3. Aggregation

Aggregation algorithms replace a set of constraints with a different formulation, typically to obtain more powerful propagation in a CP solver. CGRASS [24] replaces cliques of not-equal constraints with allDifferent constraints. IBM CP Optimizer Presolve [36] also replaces cliques of not-equal constraints with allDifferent, as well as collecting overlapping *count* expressions (that constrain the cardinality of a given value in a set of variables). IBM CP Optimizer Presolve also merges overlapping linear constraints on $\{0, 1\}$ variables, and in some cases merges arithmetic constraints with other constraint types.

Aggregation is essentially a forward-chaining approach: a pattern is identified in the set of constraints and replaced with an improved formulation such as a global constraint. Other approaches to identifying global constraints are covered in Section 2.7. Two types of aggregation are implemented in Savile Row as described in Section 5.5.

2.4. Flattening and identical common subexpression elimination

Flattening is the process of taking a nested expression and reducing the degree of nesting by replacing a subexpression with a new variable. For example, given the product $X \times (Y + Z)$ and a target solver that does not allow sums inside products, the flattening process will add a new variable *aux*, replace the product with the new expression $X \times aux$ and add a new constraint $aux = Y + Z$. We say that $X \times (Y + Z)$ is *flattened* to $X \times aux$ and that $Y + Z$ is *extracted*.

Common sub-expression elimination (CSE) is widely used in compilers to avoid repeated computation [15]. CSE was applied in the context of finite-domain constraint languages by Rendl [60,59]. CGRASS [24] could also perform CSE through the combination of its *introduce* and *eliminate* rules. In its simplest form, which we call "Identical CSE", CSE takes two or more syntactically identical sub-expressions that must be flattened, and flattens them all using the same auxiliary variable. This reduces both the number of constraints and auxiliary variables. Importantly, Identical CSE can reduce the search space dramatically [60,59] by linking different constraints together thus strengthening constraint propagation in a conventional CP solver.

The SMT solver CVC4 merges identical subtrees in its abstract syntax tree [4]. It is not clear whether this affects the search or is simply a memory saving feature.

The algorithm introduced by Rendl [59] and later extended by Stuckey and Tack [65] performs CSE during flattening. The algorithm maintains a hash table that maps each expression extracted so far to the auxiliary variable created for it.

When extracting an expression E, the algorithm looks up E in the hash table and (if present) uses the auxiliary variable in the hash table rather than creating a new auxiliary. Rendl's algorithm has the advantage of being a simple addition to the flattening process, however there are two disadvantages: the order in which CSEs are extracted is fixed to the order of flattening, and the candidate expressions for CSE must be the expressions that are extracted by flattening. For example, most constraint solvers can accept inequality constraints without flattening so Rendl's algorithm would miss the common subexpression $x+y+z$ in the constraints $x+y+z \leq 10$ and $x+y+z \geq 11$.

In this paper we present a new algorithm for Identical CSE in Section 5.7 that is independent of flattening thus avoiding both disadvantages of Rendl's algorithm. For example the new algorithm is able to extract $x+y+z$ from $x+y+z \leq 10$ and $x+y+z \geq 11$ and allow constraint solvers to immediately deduce the contradiction.

Stuckey and Tack describe CSE in the constraint language MiniZinc 2 that has user-defined functions and local decision variables [65]. Function call sites must be syntactically identical to trigger CSE, therefore it is a form of Identical CSE, essentially using the same algorithm as Rendl. The current version of Essence Prime does not have user-defined functions.

2.5. Normalisation and active CSE

One way CSE has been fruitfully extended is by matching subexpressions that are not syntactically identical [60]. This is achieved in two ways. The first is *normalisation*, where prior to CSE the expression tree is converted to a normal form. This converts some semantically equivalent expressions (such as $C = B + A + 1 - 1$ and $A + B = C$) to syntactically identical expressions. The second is *Active CSE*, where two expressions A and B may be matched if they are identical after some transformation is applied to one. For example, Active CSE can match $A < B$ with $A \geq B$ by a simple negation, replacing one with an auxiliary variable a and the other with $\neg a$. Rendl's algorithm described above is easily extended to Active CSE [59]. When an expression E is extracted by flattening, the extended algorithm looks up E and each transformation of E in the hash table. When a transformation of E is present, the inverse transformation is applied to the auxiliary variable. For example, suppose the transformation is to negate E then apply one of De Morgan's laws, the inverse is (also) to negate the auxiliary variable. Rendl's Active CSE algorithm has the same problems as the Identical CSE algorithm described above. We present a different algorithm for Active CSE in Section 5.8 and relate it to Rendl's work.

2.6. Associative–commutative CSE

One of the forms of CSE we investigate is *Associative–Commutative* CSE (AC-CSE), which exploits the properties of associativity and commutativity of binary operators (e.g. $+$ and $\times$). Expressions containing these operators can be rearranged to reveal common subexpressions. As an example, take the following two constraints over four variables, each with domain $\{0 \dots 10\}$: $w+x+y+z = 6$, $z+y+w = 5$. Conventional constraint propagation (bounds($\mathbb{R}$) consistency [13] or GAC [61] applied to both constraints) will not reveal the fact that $x = 1$. AC-CSE can extract $w+y+z$ and replace it with an auxiliary variable a to give the following three constraints: $x+a = 6$, $a = 5$, $a = w+y+z$. Performing constraint propagation (e.g. bounds($\mathbb{R}$) consistency [13]) on this set will assign x to 1.

An Associative–Commutative Common Subexpression (AC-CS) of a set of associative and commutative (AC) expressions is a set of at least two terms that appear in each of the AC expressions. In the example above, the terms w, y, z appear in both the original sum constraints, hence $\{w, y, z\}$ is an AC-CS of the two sum constraints.

A simple normalisation step, such as sorting the terms in the AC expressions, followed by examining contiguous subsequences of terms within AC expressions, can reveal some but not all of the available AC-CSs. More is necessary to find AC-CSs in general. The largest contiguous subsequence in the example above (sorted alphabetically) is $y+z$, so this approach would miss the maximal AC-CS $w+y+z$.

Araya, Neveu and Trombettoni [2] exploited common subexpressions among $+$ and $\times$ expressions in the context of numerical CSP. They proposed two algorithms named I-CSE and I-CSE-NC. Both algorithms apply a form of AC-CSE prior to flattening as a separate operation. The first pass of both algorithms is to transform the abstract syntax tree into a directed acyclic graph where identical subexpressions are represented once. The second step is to intersect each pair of sums and pair of products to create a set of candidate AC-CSs. As we will see in Section 4.3 other AC-CSs (generated by the intersection of three or more sums) can also be useful, but I-CSE and I-CSE-NC will never extract them. Later passes extract the AC-CSs from the original expressions.

Araya et al. defined two AC-CSs f_1 and f_2 to be *in conflict* if $f_1 \cap f_2 \neq \emptyset$, $f_1 \not\subseteq f_2$ and $f_2 \not\subseteq f_1$. Two AC-CSs in conflict cannot both be extracted from the same expression. When a set of AC-CSs in conflict are subsets of the same original expression s, then I-CSE copies s a sufficient number of times to extract each of the AC-CSs from at least one copy. I-CSE-NC (for No Conflicts) does not copy s, it simply extracts a single maximal subset of the candidate AC-CSs from s. Consider the following example:

$$v+w+x+y = 0, \quad v+w+x+z = 0, \quad v+w+y+z = 0$$

In this example I-CSE would extract three AC-CSs: $v+w+x$, $v+w+y$ and $v+w+z$. I-CSE would duplicate each of the original constraints resulting in six constraints and three further constraints to define the auxiliary variables. I-CSE-NC can extract only one AC-CS. Suppose it extracts $v+w+x$, then at this point $v+w+y$ and $v+w+z$ cease to be AC-CSs:

$$aux + y = 0,\ aux + z = 0,\ v+w+y+z = 0,\ aux = v+w+x$$

I-CSE-NC can only extract AC-CSs from the original expressions, so fails to exploit the AC-CS $v + w$. Even on this small example I-CSE increases the size of the model substantially. I-CSE-NC has not, but it has missed a potentially useful AC-CS and has not linked $v + w + y + z$ to the other two sums. I-CSE and I-CSE-NC are both compared with our algorithm in Section 5.11. We implement the algorithms exactly as described in Section 4 of Araya et al. [2]. Both I-CSE and I-CSE-NC only extract AC-CSs from the original expressions, not from other AC-CSs.

Bessière, Paparrizou and Stergiou [9] define two new consistencies, rBC2 and rBCall, both stronger than bound consistency [13]. Where bound consistency reasons with each constraint separately, rBC2 and rBCall consider multiple constraints together. Two reformulations are proposed that approximate rBC2 and rBCall for linear constraints: rBC2-Y and rBCall-Y. Whenever a pair of linear constraints share more than one variable and the coefficients are in the same proportion in both constraints, rBC2-Y extracts an AC-CS. For example, $2x + 3y + z = 5$ and $4x + 6y + w = 5$ share variables x and y, and the coefficients of x and y are in the same proportion in both constraints. When a constraint has multiple AC-CSs it is copied once for each. Where rBC2-Y links pairs of constraints, rBCall-Y links sets of constraints that share two or more variables with proportional coefficients. In other respects they are the same. Both are defined for a flat set of linear (in)equality constraints.

In contrast to rBC2-Y and rBCall-Y, I-CSE requires coefficients to be identical to extract an AC-CS. However I-CSE can handle nested expressions in an AST and it packs AC-CSs into a minimal set of copies of the original expression.

In Section 4 we describe our approach to AC-CSE and set it in context with I-CSE, rBC2-Y and rBCall-Y.

2.7. Other related work

Charnley, Colton and Miguel [12] proposed an automated method of generating implied constraints for a problem class. They first solve small instances of the class and use machine learning to generate hypothetical implied constraints, and then use a theorem prover to test each hypothesis and those proven correct are added to the model. As a time-intensive method that works on the problem class it is quite different from our approach.

Leo et al. [38] proposed a method to suggest global constraints to replace groups of constraints in a model. The method generates sample solutions of each candidate group of constraints and suggests global constraints from its library. This method is also time-intensive, requires human verification and is applied to a problem class.

Bessière et al. [8] proposed an approach to learning implied parametric global constraints such as the global cardinality constraint (GCC) for problem instances. While promising results are given, the approach requires some human insight and is also likely to be much more time-consuming than syntactic operations such as aggregation and CSE.

Symmetry and dominance breaking constraints may be generated at the class level using the methods of Mears et al. [42,41]. A dominance breaking constraint rules out sub-optimal solutions of an optimisation problem. Symmetry and dominance breaking constraints are complementary to our work since they can enable other reformulations (as in the motivating example in Section 1.2).

In SAT, research on preprocessing [66,17], and inprocessing [34] interleaves the reformulation of a SAT model with solving. An example reformulation is "self-subsuming resolution" proposed by Eén and Biere [17]. Given the clauses $C_1 = \{x, a, b\}$, and $C_2 = \{\bar{x}, a\}$, the resolvent is $C'_1 = \{a, b\}$. Since C'_1 subsumes C_1 we can replace C_1 with C'_1. Common subexpression elimination has been used in SAT by identifying common subclauses and replacing them by a new auxiliary variable and clauses equating the variable with the common subclause [70]. Preprocessing and inprocessing are highly effective in extending the reach of SAT solvers, however they necessarily operate on the lowest level representation of the problem, the raw SAT clauses, where higher-level patterns are difficult and/or costly to identify. In Savile Row, by contrast, a single common subexpression elimination can eliminate many SAT variables and clauses in the encoded problem.

In a recent wide-ranging review Achterberg et al. [1] demonstrate the central importance of preprocessing techniques to the performance of modern MIP solvers. Similarly to Savile Row, MIP preprocessing consists of the application of a collection of relatively inexpensive procedures that together work to strengthen the model of a problem instance. These procedures are tailored to the linear inequalities that form the MIP input language, as opposed to the richer constraint language considered by Savile Row, removing redundant inequalities, replacing inequalities with stronger equivalents, strengthening coefficients, and strengthening the bounds on variables.

Constraint Handling Rules (CHR) [26] is a declarative, rule-based, general-purpose programming language originally designed to write constraint solvers [27]. The rule-based formalism of CHR is a natural way to encode transformations similar to those made by Savile Row. However, we are not aware of work employing CHR to reformulate problem instances prior to search by a constraint or SAT solver. Cadmium [16] is an associative, commutative term rewriting system, inspired by CHR, designed to map the Zinc [3] language to solvers. To the best of our knowledge, in the transformation of Zinc associativity and commutativity are used not for common-subexpression elimination, but solely for matching local expressions. For example, a rule that matched $a \vee \neg a$ would also match $a \vee b \vee \neg a$, because $\vee$ is associative and commutative.

3. Architecture of Savile Row

Savile Row is a multi-pass term rewriting system. The frontend reads a model in the Essence Prime language into an abstract syntax tree (AST), along with a parameter file giving the values of the problem class parameters. Several backends target a range of solvers. The system has many passes, some of which are always performed, others are performed when

the choice of backend requires it, and still others are optional reformulations. In this section we give an overview of the essential parts of the system, excluding optional reformulation passes described in Section 5. We refer to the entire process of transforming a model into input for a solver as the *tailoring* process [59].

3.1. First steps of the tailoring process

The first part of the tailoring process puts the problem instance into a form suitable for the later optional reformulation passes. The details are unimportant for this paper but the key points are as follows.

1. The relational semantics [25] is implemented by replacing all partial functions (e.g. division) with total functions [49].
2. Problem class parameters and other constants are substituted into the model.
3. All quantifiers and matrix comprehensions are unrolled.
4. Matrices of decision variables (defined by `find` statements) are separated into individual decision variables. This enables domain filtering and unification of equal variables.

3.2. Bounds of expressions

Each numerical expression type has a method to estimate the lower and upper bound of its value. For non-atomic expressions the method recursively finds the bounds of each of its subexpressions then calculates the bounds assuming each subexpression is independent. For example the lower bound of a sum $1 + x + y$ is the sum of the lower bounds of 1, x and y. The method for variable identifiers takes the lower and upper bounds of the domain (from the quantifier, comprehension or the global symbol table that defines the variable). Bounds calculated in this way for an expression e are denoted $\lfloor e \rfloor$ and $\lceil e \rceil$. The estimated bounds are required to contain the true value of e in all solutions.

3.3. Simplifiers

Each expression type implemented in Savile Row has a *simplifier*, which is a function that examines an expression and may rewrite it to a semantically equivalent simpler expression. Simplifiers may examine the subexpressions in any way, including looking up the domains of variables and calculating bounds of subexpressions. As a minimum requirement the simplifier must evaluate the expression (i.e. replace it with a constant of the appropriate type) if the expression directly contains only constants. For example, the allDifferent constraint allDiff($[1, 3, 4]$) contains a constant matrix, and since all the elements of the matrix are different it must be rewritten to *true*. On the other hand, allDiff($[1, 2, x]$) where x is a decision variable with domain $\{3 \ldots 10\}$ is rewritten to *true* in the current version of Savile Row, but this is not a requirement.

The simplifier for the expression type allDiff (that contains a one-dimensional matrix) performs the following rewrite steps:

- If the matrix is unknown (e.g., a matrix comprehension) then do not rewrite the allDiff, otherwise continue.
- If the matrix is of size 0 or 1 then rewrite the allDiff to *true*.
- If any pair of expressions in the matrix are numerically equal or their ASTs are identical then rewrite to *false*.
- For each expression in the matrix, find its bounds or (if it is a variable) look up its domain. If any constant c in the matrix is outside the bounds or domain of all other expressions then rewrite to a shorter allDiff that does not contain c.

Notice that the simplifier does nothing if the allDiff contains a matrix comprehension or a function such as `flatten`. Quantifiers and comprehensions are unrolled early in the tailoring process (Section 3.1), and functions are evaluated as soon as their contents have the appropriate properties. For example, the `flatten` function is evaluated when the matrix contained in it is iterable in all dimensions. Once `flatten` has been evaluated, the allDiff simplifier will execute immediately.

Simplifiers are applied to the AST in a bottom-up order. This allows each simplifier to assume that the children of the current AST node have already been simplified. Simplifiers are run to quiescence, as follows: suppose some AST node n_1 is replaced with n_2 by a simplifier. The tree rooted at node n_2 is then traversed in bottom-up order and n_2 or subtrees of it may be replaced again. Once the entire tree traversal has completed, the simplifier for each node has been executed and no simplifier can perform any further rewrites.

Simplifiers have a central role in Savile Row and cannot be switched off. Other passes assume that the simplifiers have been run to quiescence. Whenever one pass makes a change to the AST, the simplifiers are run to quiescence before the next pass.

3.3.1. Negation normal form

Simplifiers for the logical connectives $\wedge$, $\vee$ and $\neg$ are responsible for establishing a type of negation normal form by applying De Morgan's laws to push negation towards the leaves of the AST, and by removing double negation. Implication $A \rightarrow B$ is rewritten to $\neg A \vee B$. In some cases a negated constraint $\neg A$ can be replaced by B where B is no more complex than A (e.g. $\neg(A = B)$ is rewritten to $A \neq B$). The numerical comparison operators $=$, $\neq$, $\leq$, $<$ and also `table` (where the satisfying assignments of the constraint are explicitly listed) and `negativetable` (the negation of `table`) are all rewritten when negated.

Algorithm 1 X-CSE(AST, ST, $\diamond$).

Require: AST: Abstract syntax tree representing the model
Require: ST: Symbol table containing decision variables
Require: $\diamond$: The associative and commutative operator
1: *newcons* $\leftarrow$ empty list {Collect new constraints}
2: *map* $\leftarrow$ empty hash table mapping pairs of expressions to lists
3: populateMapAC(AST, *map*, $\diamond$)
4: **while** *map* not empty **do**
5: *pairexp* $\leftarrow$ heuristic(*map*)
6: *ls* $\leftarrow$ *map*(*pairexp*) {*ls* is a list of $\diamond$ AST nodes}
7: delete *map*(*pairexp*)
8: *ls* $\leftarrow$ filter(isAttached, *ls*) {Remove $\diamond$ AST nodes no longer contained in AST or *newcons*}
9: **if** length(*ls*) > 1 **then**
10: *commonset* $\leftarrow$ *ls*[1] $\cap$ *ls*[2] $\cap \cdots \cap$ *ls*[length(*ls*)]
11: *e* $\leftarrow$ fold($\diamond$, *commonset*)
12: *dom* $\leftarrow$ domain(*e*)
13: *aux* $\leftarrow$ ST.newAuxVar(*dom*)
14: *newc* $\leftarrow$ (*e* $=$ *aux*) {New constraint defining *aux*}
15: *newcons*.append(*newc*)
16: **for all** $a \in ls$ **do**
17: *newe* $\leftarrow$ fold($\diamond$, $(a \setminus commonset) \cup \{aux\}$)
18: Replace *a* with *newe* within AST or *newcons*
19: populateMapAC(*newe*, *map*, $\diamond$)
20: populateMapAC(*newc*, *map*, $\diamond$)
21: AST $\leftarrow$ AST $\wedge$ fold($\wedge$, *newcons*)

3.4. General flattening

Once all optional reformulation passes have completed, any remaining nested expressions where the target solver does not support the nesting (e.g. a sum containing a product) are flattened by extracting the inner expression e to a new auxiliary variable a. The domain of a is $\{\lfloor e \rfloor \dots \lceil e \rceil\}$ unless extended domain filtering is enabled (Section 5.4 below). Extended domain filtering may provide a smaller domain for a.

3.5. Encoding to SAT

Our goal is to investigate whether reformulations performed on a constraint problem instance are beneficial when it is solved by encoding to SAT and using a state-of-the-art SAT solver. To achieve this we need to ensure that the baseline encoding to SAT is sensible. We have used standard encodings from the literature such as the order encoding for sums [67] and support encoding [29] for binary constraints. Also we do not attempt to encode all constraints in the language: several constraint types are decomposed before encoding to SAT. Further details are given elsewhere [48].

4. The X-CSE algorithm for associative commutative CSE

We described existing approaches to common subexpression elimination for associative and commutative operators in Section 2.6. One of the main contributions of this paper is a new algorithm, which we call X-CSE, to eliminate AC-CSEs. For any AC operator $\diamond$ (for example, sum, product or disjunction), the goal of X-CSE is to find common sets containing two or more expressions that are contained in more than one $\diamond$ expression. These are extracted to a new variable, potentially improving constraint reasoning in the solver and reducing the size of the model.

The X-CSE algorithm uses a hash table *map* from *pairs* of expressions $\{a, b\}$ to a list of the $\diamond$ expressions that contain both a and b. Algorithm 2 (populateMapAC) takes a reference to an AST node and explores the tree, populating *map* for each $\diamond$ expression. If the $\diamond$ expression contains two copies of a, *map* will contain $\{a, a\}$.

Algorithm 1 (X-CSE) takes a reference to the AST representing all constraints, a reference to the global symbol table, and the AC operator $\diamond$. After initialising data structures it calls populateMapAC with the entire AST. Following that it enters the main loop on line 4. On line 5 one pair is selected from *map* according to a heuristic. If the pair occurs in more than one $\diamond$ expression then there must exist an AC-CS including that pair. Lines 10–20 find an AC-CS and extract it from all the relevant expressions. The algorithm includes as many $\diamond$ expressions as possible to maximise the effect of extracting the AC-CS. Line 10 intersects all $\diamond$ expressions containing the pair. A new $\diamond$ expression for the AC-CS is constructed, and an auxiliary variable is created. On line 14 a constraint is created to define the auxiliary variable. Each $\diamond$ expression containing the AC-CS is replaced. At this point, lines 19 and 20 update *map* to include all the newly created expressions, allowing X-CSE to extract further AC-CSs from the new expressions. Some references to removed $\diamond$ expressions will remain in *map*; these will be filtered out on line 8.

The implementation of X-CSE is optimised by using a second hash table recording the global number of occurrences of every expression directly contained in a $\diamond$ expression. If an expression e_1 occurs only once, then it cannot take part in an AC-CS and no pairs $\{e_1, e_2\}$ are stored in *map*.

We apply X-CSE to sums, products, conjunctions and disjunctions in our experiment in Section 5.9 below.

Algorithm 2 populateMapAC(A, *map*, $\diamond$).

Require: A: Reference to an AST
Require: *map*: Hash table mapping pairs of expressions to lists
Require: $\diamond$: The associative and commutative operator
1: **if** A is expression of $\diamond$ **then**
2: **for all** $\{e_1, e_2\} \subseteq A$ **do**
3: Add A to list *map*$[\{e_1, e_2\}]$
4: **for all** *child* $\in A$.Children() **do**
5: populateMapAC(*child*, *map*, $\diamond$)

4.1. Heuristics

X-CSE chooses the next pair to process by calling a heuristic on line 5. Recall (from Section 2.6) that two AC-CSs in conflict cannot both be extracted from the same expression, therefore a choice is sometimes required. We experimented informally with eight heuristics. There are four basic heuristics: most occurrences (i.e. select the pair that leads to the longest list *ls* after line 8 of X-CSE), fewest occurrences, largest AC-CS and smallest AC-CS. In some cases there exists a pair such that its corresponding AC-CS can be extracted without preventing any other AC-CS. We call these *non-blocking pairs* and it may be helpful to process them first. We created four more heuristics that select non-blocking pairs first, then fall back to one of the four basic heuristics. We found no clear winner among the eight heuristics. We use the 'most occurrences' heuristic (without prioritising non-blocking pairs) throughout the rest of this paper because it is cheap to compute and often performs well.

4.2. Complexity analysis

In this analysis we will use n for the number of $\diamond$ expressions, k for the length of the longest $\diamond$ expression, d as the depth of the deepest $\diamond$ expression in the AST, and S as the number of nodes in the AST. Central to the complexity analysis is the observation that at most $k-1$ AC-CSs may be extracted from one $\diamond$ by X-CSE. The smallest AC-CS is size two, and extracting this replaces a size two term with a size one term (i.e. the replacement auxiliary variable). If the original expression is size k, we thus find one AC-CS and now have a size $k-1$ expression. Iterating shows that at most $k-1$ AC-CSs may be extracted from one $\diamond$ expression by X-CSE. This gives us a global limit of $O(nk)$ AC-CS extractions.

To populate *map*, populateMapAC traverses the AST with S nodes, and for each $\diamond$ expression e it inserts a reference to e in $O(k^2)$ lists within *map*. Assuming hash table operations are $O(1)$, populateMapAC takes $O(S + nk^2)$ time.[1]

X-CSE then enters a loop that continues until *map* is empty. Each iteration of the loop is as follows. We assume the heuristic takes $O(1)$ time.[2] For the given pair, its list *ls* has at most n elements. Note that if the pair occurs more than once in an expression it might be entered into *ls* multiple times: to keep the list at size n, when inserting an expression e into *ls* we can check the last element of *ls*: if it is equal to e, we do not insert e for a second time. The list *ls* is filtered in $O(nd)$ time. If the list has length two or greater, then we extract an AC-CS. For the following we assume that an AC expression is represented by a set data structure with $O(1)$ lookup, insertion and removal. Creating *commonset* on line 10 takes $O(nk)$ time. Computing the domain and creating the auxiliary variable and the new constraint can be done in $O(k)$ time. The algorithm then replaces *commonset* in each *ls* expression in $O(nk)$ time. Re-populating *map* (on lines 19 and 20) takes $O(S + nk^2)$ because the updated AC expressions can contain the entire AST. Therefore the entire cost of extracting one AC-CS is $O(S + nk^2 + nd)$, and the total cost of X-CSE is $O(nkS + n^2k^3 + n^2kd)$.

While the complexity may seem high, the algorithm scales with the number of AC-CSs it is able to exploit, therefore it is relatively quick when there are few or no AC-CSs, and it takes more time when there is greater potential benefit.

4.3. Comparison with related algorithms

X-CSE differs from the existing algorithms I-CSE(-NC) in that it can extract AC-CSs that are intersections of more than two expressions and AC-CSs containing auxiliary variables (from earlier steps). Thus it has a larger palette of AC-CSs to choose from. Consider the example from Section 2.6: $v + w + x + y = 0$, $v + w + x + z = 0$, $v + w + y + z = 0$. I-CSE(-NC) would have a palette of three AC-CSs: $v + w + x$, $v + w + y$ and $v + w + z$. X-CSE would first extract $v + w$ from all three sums:

$$a = v + w, \quad a + x + y = 0, \quad a + x + z = 0, \quad a + y + z = 0$$

Second, X-CSE would extract any one of $a + x$, $a + y$ or $a + z$, as follows. This second step is not possible in I-CSE(-NC).

$$a = v + w, \quad b = a + x, \quad b + y = 0, \quad b + z = 0, \quad a + y + z = 0$$

[1] This is correct if all expressions to be hashed are size $O(1)$ and computing the hash code is linear. If either assumption is invalid then an additional factor h is necessary for the time to hash an expression.

[2] As an example of an $O(1)$ heuristic we could maintain a doubly linked list of keys in *map* and have the heuristic simply remove and return the first element of the list.

This result is clearly better than I-CSE-NC that extracted only $v + w + x$ and thus did not connect the third constraint to the other two. I-CSE produced nine constraints on this example. I-CSE can drastically increase the size of the instance where X-CSE often makes it more compact. The two are compared experimentally in Section 5.11.

On this example, I-CSE misses the AC-CS $v + w$ that connects all three of the original constraints. We propose I-CSE-Iter that iterates I-CSE. I-CSE-Iter first calls I-CSE on the original constraints, then repeatedly calls I-CSE on the AC-CSs extracted by the previous call until no more AC-CSs are found. The second iteration of I-CSE-Iter would find $v + w$, and indeed it can extract any AC-CS that X-CSE can extract, solving one potential disadvantage of I-CSE. However, iterating I-CSE may further increase the size of the problem instance. We evaluate I-CSE-Iter in Section 5.11.

Both rBC2-Y and rBCall-Y (Section 2.6) are able to extract AC-CSs of sums where the coefficients are not identical but in the same proportion in each sum. They are much closer to I-CSE (rBC2-Y) and I-CSE-Iter (rBCall-Y) than to X-CSE because copies of the original expressions are made to allow extraction of conflicting AC-CSs. However rBC2-Y and rBCall-Y simply copy the expressions once for each AC-CS rather than packing the AC-CSs into the minimal number of copies. We do not compare X-CSE with rBC2-Y and rBCall-Y in our experiments, but we have extended X-CSE to capture a common case of proportional coefficients in Section 5.12 below.

4.4. Implied constraints for AC-CSE

A further step to promote the identification of AC-CSs is in reformulating a model to add implied constraints consisting of AC expressions. Savile Row creates implied sum constraints from allDifferent and global cardinality (GCC) constraints. This is done by finding a lower and upper bound on the sum of the variables in the allDifferent or GCC (lb and ub resp.), then adding either $\sum \geq lb$ and $\sum \leq ub$ (when $lb \neq ub$) or $\sum = lb = ub$ where $\sum$ is the sum of the variables in scope of the original constraint (except cardinality variables in GCC). The implied constraints are simple examples of *finite-domain cuts* as investigated by Bergman and Hooker [6].

To find the value of ub (lb) for a given allDifferent or GCC constraint, we find an assignment to all the relevant variables such that each variable takes a value from its domain, the sum of the assignment is maximized (minimized), and the number of occurrences of each value is within the interval permitted for the value (for AllDifferent, the interval is $\{0 \dots 1\}$ for every value). The assignment is found using the minimum cost maximum flow formulation described by Régin [57]. If no assignment exists with the required properties then the constraint is replaced with *false*. For example, given constraint allDifferent(x, y, z) where all variables have domain $\{1, 3, 5, 7\}$, Savile Row adds constraints $x + y + z \geq 1 + 3 + 5$ and $x + y + z \leq 3 + 5 + 7$.

The implied constraints are only added when AC-CSE is switched on. They are added directly before AC-CSE, and any implied constraint that is unchanged by AC-CSE is removed afterwards.

5. Reformulations in Savile Row

In this section we describe a range of reformulations, give algorithms to perform them, and evaluate them on a benchmark set. We evaluate each of the reformulations separately. Their inter-dependence raises a problem: we cannot try every subset. We have opted for a cumulative set of experiments, starting with the bare-bones configuration of Savile Row and in each experiment adding one reformulation. If it is successful, it is retained for all the subsequent experiments.

5.1. Benchmark set

The benchmark set is the set of example models and parameter files included with Savile Row 1.6.5 (available from http://savilerow.cs.st-andrews.ac.uk/). There are 50 problem classes with 596 instances in total. The set contains a mixture of satisfaction and optimisation problems. Models are described as required in the sections below.

5.2. Experimental context

In our experiments we target the conventional propagation-based CP solver Minion 1.8 (64-bit), with a static variable and value ordering. The static search order means that any reduction in search tree size must be caused by improved propagation, so we directly see the effects of improving constraint formulation. We switch on Minion's option to perform singleton bound consistency (as described in Section 2.2) directly before starting search. We also target the SAT solver Lingeling [34] which was winner of the Sequential, Application SAT+UNSAT track of the SAT competition 2014.[3] We used Savile Row 1.6.5, executed in the OpenJDK Java 1.7.0_141 VM with the 64-bit server VM. Each reported timing is a median of 5 runs when Minion is the target solver, and 10 runs (with 10 distinct random seeds $1, \dots, 10$) when targeting Lingeling. Minion always performs the same search given the same input but there is some variation in run time caused by the environment. The search performed by Lingeling may be different for two random seeds, so the larger sample of 10 runs

[3] Lingeling version ayv 86bf266b9332599f1b876e28a02fe8427aeaa2db.

Table 1
Configurations of Savile Row used in the experiments below.

Config name	Description
Simp	Simplifiers are enabled. All other optimisations are disabled.
VarUnif	Simp plus variable unification.
DomFilt	VarUnif plus domain filtering.
EDomFilt	DomFilt plus extended domain filtering.
Aggreg	DomFilt plus aggregation.
IdentCSE	Aggreg plus Identical CSE.
ActiveCSE	IdentCSE plus Active CSE.
X-CSE	IdentCSE plus AC-CSE implemented by X-CSE.
X-CSE-Alone	Simp plus AC-CSE implemented by X-CSE.
I-CSE	IdentCSE plus AC-CSE implemented by I-CSE.
I-CSE-NC	IdentCSE plus AC-CSE implemented by I-CSE-NC.
I-CSE-Iter	IdentCSE plus AC-CSE implemented by I-CSE-Iter.
ActiveXCSE	IdentCSE plus Active AC-CSE implemented by Active X-CSE.

is to mitigate the greater variation in the run times. Both solvers and Savile Row are single-threaded. Experiments were performed with 32 processes in parallel on a 32-core AMD Opteron 6272 at 2.1 GHz with 256 GB RAM.

The reported timings for Savile Row *include* time spent running Minion to do domain filtering, and do not include time spent running the target solver (whether it is Minion or Lingeling). In some cases we report timings of the target solver only, and in other cases total time (which is simply Savile Row time plus target solver time). When Minion is the target solver, a time limit of 30 minutes is applied to the total time. With Lingeling a time limit of 1 hour is applied. Lingeling is able to solve many more of the instances than Minion so this allows us to use a longer time limit.

There are 13 configurations of Savile Row used in the experiments, summarised in Table 1. The table refers to the reformulations described in Sections 5.4 to 5.12 below.

5.3. Statistical analysis

To compare two configurations A and B of Savile Row, we first take the median of the total time for each instance and each configuration. The total time is the time taken by Savile Row (including any calls to a solver for preprocessing) plus time taken by the backend solver. We chose the median because it is less affected by outliers than the mean. Instances where either configuration timed out are discarded. For the remaining instances, we take the quotient of the two medians ($\frac{B}{A}$). Finally we take the geometric mean of the set of quotients to obtain a single statistic comparing the two configurations. A is considered better than B iff the geometric mean is greater than 1.

We chose the geometric mean because it is more appropriate for quotient values than the arithmetic mean, being based on product rather than sum. Suppose one instance was twice as fast with configuration A, and another was twice as fast with B. The two quotients would be 2 and 0.5, and their geometric mean would be 1. The arithmetic mean would be 1.25 which gives the misleading impression that A is faster.

If the geometric mean is close to 1, it may not be clear whether there is a significant improvement between the two configurations. To test for a significant improvement using a traditional statistical test, we would need a non-parametric one-tailed test. However, non-parametric tests are inappropriate as they fail to take account of degree of difference. Suppose (when enabling some feature of Savile Row) 10% of instances are 100 times faster, and the other 90% are 1% slower. Non-parametric tests would be unlikely to reach significance or may even indicate the feature had a negative effect. Instead of using a traditional non-parametric test we compute a confidence interval for the geometric mean by bootstrapping. There are 596 benchmark instances and we take 100,000 bootstrap samples (with replacement) of size 596 from the set of benchmark instances. For each bootstrap sample we compute the geometric mean and take the quantiles at 2.5% and 97.5% to give a 95% confidence interval. If the interval contains 1 then we have not reached the significance threshold. Otherwise, the difference is statistically significant. The location of the interval (above or below 1) indicates which configuration is faster.

For example, when comparing X-CSE to IdentCSE (with the CP solver) the geometric mean is 1.492, and the bootstrap confidence interval is [1.326, 1.693]. Fig. 2 plots the distribution of the bootstrap samples. X-CSE has a large benefit for some benchmark instances and makes almost no difference to others. Despite the highly asymmetric and clustered quotient data, the histogram has a smooth (possibly slightly skewed) bell-curve shape. This gives us confidence that 100,000 samples is sufficient.

5.4. Experiment one: variable unification and domain filtering

Variable unification is implemented in several modelling tools as described in Section 2.1. In Savile Row variable unification is implemented as an addition to the simplifier mechanism (Section 3.3). During a simplifier pass, when an expression $x = y$ or $x \leftrightarrow y$ is found in the top level conjunction, one variable is chosen to be removed. Supposing y will be removed, the new domain of x will be the intersection of the two domains and y will be replaced by x throughout. Similarly when an expression $x = c$ or $x \leftrightarrow c$ is found (with constant c) or the domain of x is $\{c\}$, x is replaced by c throughout. Also for

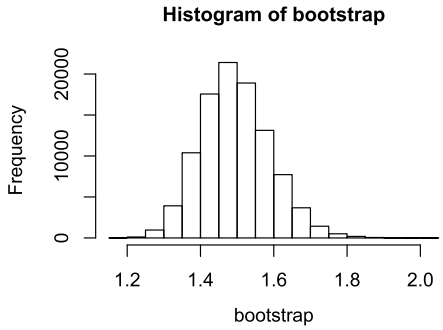

Fig. 2. Histogram of the geometric means of the bootstrap samples when comparing X-CSE to IdentCSE.

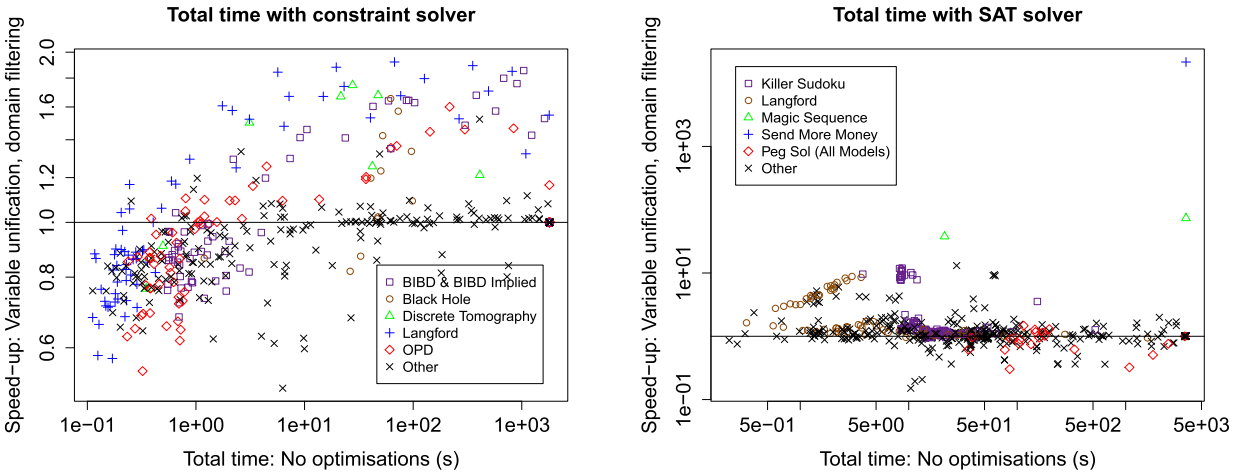

Fig. 3. DomFilt vs Simp. Both plots show total time (i.e. solver time plus Savile Row time). On the left the solver is Minion and on the right it is SAT (Lingeling). In each case, the y axis is the speed up quotient caused by the optimisation, so points above 1 on the y axis exhibit a speed improvement. The x axes are time (in seconds) for the variant without the optimisation. Both axes are logarithmic.

a Boolean variable x, if x or $\neg x$ appears in the top level conjunction then x is assigned. When variable unification is used, it is applied during all simplifier passes. In each pass, variable unification and simplifiers are applied until a fixed point is reached.

Domain filtering is an established preprocessing technique as described in Section 2.2. In Savile Row domain filtering is implemented by tailoring the problem instance for output to Minion and using Minion's standard propagation combined with singleton consistency [7] applied to the upper and lower bounds only (SACBounds). Subsequently the tailoring process is performed a second time, making use of the filtered variable domains. The second tailoring process may target a different solver or class of solvers, but is otherwise identical to the first tailoring process. MiniZinc employs a similar approach with two tailoring processes [39] using the standard propagation of the Gecode solver [28]. The main difference is that we use the more powerful and time-consuming SACBounds.

In this paper we consider two versions of domain filtering. The first (*standard* domain filtering) is to apply domain filtering to variables defined by `find` statements only. Variables defined by `find` statements always have the same name in the first and second tailoring process, therefore it is straightforward to apply the filtered domains in the second tailoring process. This is the version used in our previous work [47,50]. *Extended* domain filtering (EDF) transfers filtered domains of auxiliary variables that may have different names in the first and second tailoring process. This is done by linking each auxiliary variable to an expression that it represents. The expression acts as a canonical name that remains the same in the first and second tailoring process. MiniZinc implements extended domain filtering using a different canonical name [39]. The EDF algorithm is given in the supplementary material [48] with an experiment that shows no significant improvement over standard domain filtering.

Experimental results

For brevity we evaluate variable unification and standard domain filtering together in one experiment. Elsewhere we give fuller results with two separate experiments [48]. In Fig. 3 we compare standard domain filtering plus variable unification to the basic configuration with neither option (DomFilt vs Simp). With the CP solver there is an improving trend: for easy instances the overhead of performing the entire tailoring process twice dominates, but for harder instances the benefit can

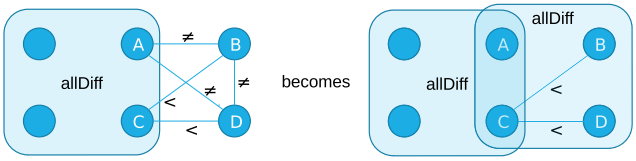

Fig. 4. Aggregation of allDifferent on arbitrary expressions A, B, C and D.

outweigh the cost. The trend is particularly clear for BIBD, Langford and OPD problem classes. Geometric mean speed-up of all instances is 0.962 with confidence interval [0.939, 0.987]. Instances where Simp took 10 seconds or more (total time) have a geometric mean speed-up of 1.176 with confidence interval [1.129, 1.226].

When using the SAT solver the effect varies considerably with some instances being solved (i.e. all variables are assigned) by domain filtering (Send More Money, Magic Sequence) and thus exhibiting large speed-ups. Some other instances such as those of Langford and Killer Sudoku are substantially improved. For 572 of 596 instances the SAT encoding has fewer variables and fewer clauses and for another 13 instances the encoding remains the same size. 11 instances exceeded the clause limit and no SAT encoding was produced. In some cases, performance is degraded and the reason is not clear. For example, on two instances of Peg Solitaire the SAT solver is more than two times slower, despite the number of SAT variables being reduced by 40% on average. However, on the full set of 596 instances the improvement is clear: the geometric mean speed-up is 1.329 with confidence interval [1.256, 1.407].

In summary, variable unification and standard domain filtering are beneficial on average for harder instances (taking over 10 seconds) or when targeting SAT. Furthermore they enable AC-CSE to substantially improve the formulation of some problem classes (as shown in Section 5.9 below). Hence both reformulations will be switched on for all subsequent experiments.

5.5. Experiment two: aggregation

Section 2.3 surveys the existing work on aggregation. In the current version of Savile Row two types of aggregation are implemented. The first collects $\neq$, $<$ and allDifferent expressions into a new allDifferent using a greedy maximal clique finding algorithm. A graph is constructed where vertices represent expressions and edges indicate that two expressions must be different. The algorithm builds a clique starting with an arbitrary edge representing a $\neq$ or $<$ constraint. If the clique has 3 or more vertices then an allDifferent is introduced and all subsumed $\neq$ constraints are replaced with *true* in the AST (the graph remains unchanged). The algorithm continues to build cliques starting with $\neq$ or $<$ constraints (excluding those that have already been included in a new allDifferent) until all such constraints have been considered. Fig. 4 gives an example. The allDifferent constraints introduced by aggregation may contain any numerical expression. Supposing the graph has n vertices and m edges, the greedy clique finding algorithm takes time $O(mn^2)$. Our results with aggregation of allDifferent in Savile Row confirm those reported for other systems that have a similar clique finding algorithm [24,36].

The second type of aggregation collects a set of `atleast` and `atmost` constraints with identical scopes into a single global cardinality constraint on the same scope, constraining the union of the sets of values of the source constraints.

The two types of aggregation are both lightweight, performing only syntactic operations with no iteration over variable domains, so the risk of slowing down tailoring is small and the potential benefit of identifying a global constraint is significant.

Experimental results

We compare the configurations Aggreg and DomFilt in Fig. 5. With the CP solver the problem classes Golomb Ruler, Graph Colouring and N-Queens 2 are most affected by aggregation (each case being aggregation of not-equal constraints to allDifferent). Aggregation improves performance of the CP solver, with two exceptions in the Graph Colouring class that take less than 1 s to solve. The largest speed-up is 231 times on the Golomb Ruler instance size 10. On Golomb Ruler, aggregation identifies the global allDifferent constraint and as a side effect removes many identical occurrences of difference expressions `ruler[j]-ruler[i]`. The geometric mean speed-up quotient (of total time) is 1.066 with confidence interval [1.012, 1.133].

The picture is different when using the SAT solver. For Graph Colouring (most instances) and Golomb Ruler, aggregation reduces the performance of the SAT solver. This indicates that the encoding of the set of allDifferent constraints can be worse than the encoding of the original set of not-equal constraints. AllDifferent is decomposed into a sum (≤ 1) constraint for each value, then a generic sum encoding is used as described in Section 3.5. For Golomb Ruler the decomposition re-introduces duplication of the expressions `ruler[j]-ruler[i]` for all $i < j$, hence aggregation is complemented by Identical CSE (as shown in the following experiment).

For Graph Colouring, aggregation affects four out of five instances, and for all four it makes the SAT encoding larger. The effect on solver efficiency is negative. A single $\neq$ constraint can be part of more than one maximal clique and hence after aggregation it may be represented in more than one allDifferent constraint. Therefore when using aggregation a single $\neq$ constraint may be represented multiple times in the SAT encoding, leading to a larger and less efficient encoding.

The geometric mean speed-up with the SAT solver is 0.983 with confidence interval [0.964, 0.999]. In summary, aggregation is beneficial for the CP solver and marginal for the SAT solver, and it will be enabled for all subsequent experiments.

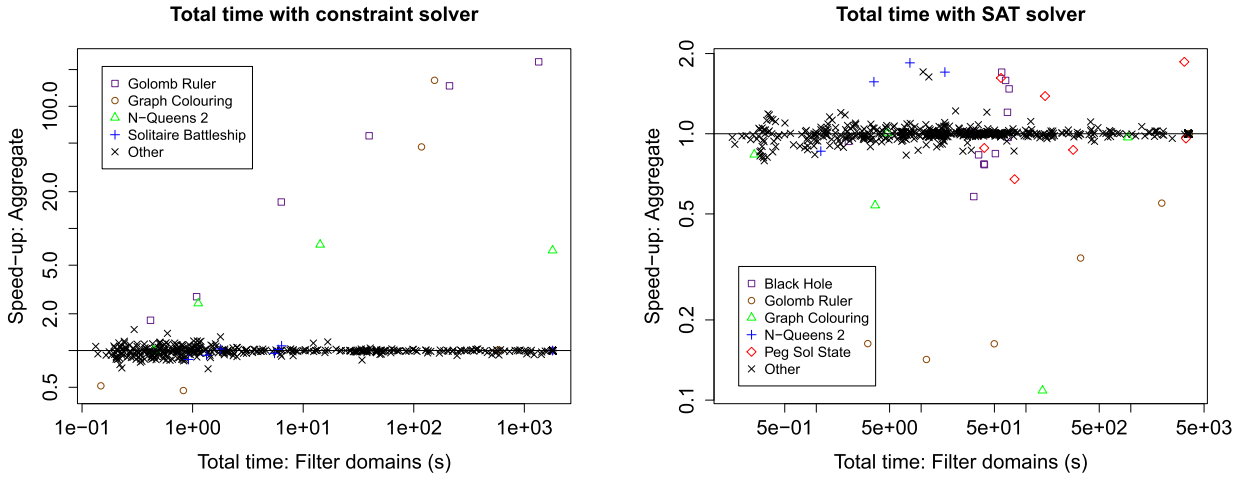

Fig. 5. Aggreg vs DomFilt total time. On the left the solver is Minion and on the right it is SAT. Axes are the same as Fig. 3.

Algorithm 3 Identical-CSE(AST, ST).

Require: AST: Abstract syntax tree representing the model
Require: ST: Symbol table containing CSP decision variables
1: *newcons* ← empty list {Collect new constraints}
2: *map* ← empty hash table mapping expressions to lists
3: populateMap(AST, *map*)
4: **for all** *key* in *map* in decreasing size order **do**
5: *ls* ← *map*(*key*) {*ls* is a list of identical AST nodes}
6: *ls* ← filter(isAttached, *ls*) {Remove AST nodes no longer contained in AST or *newcons*}
7: **if** length(*ls*) > 1 **then**
8: *e* ← head(*ls*)
9: *dom* ← domain(*e*)
10: *aux* ← ST.newAuxVar(*dom*)
11: *newc* ← (*e* = *aux*) {New constraint defining *aux*}
12: *newcons*.append(*newc*)
13: **for all** *a* ∈ *ls* **do**
14: Replace *a* with copy(*aux*) within AST or *newcons*
15: AST ← AST ∧ fold(∧, *newcons*)

5.6. Normalisation for CSE

A variety of existing CSE techniques are surveyed in Sections 2.4, 2.5 and 2.6. Normalisation [60] may be applied to extend the reach of CSE reformulations.

The simplifiers described in Section 3.3 place the AST into a limited normal form. Normalisation extends this normal form by sorting the subexpressions of all expressions where the order is unimportant. For example, $x + y + z$ and $y + z + x$ are semantically equal and normalisation will make them syntactically identical. Apart from associative–commutative operators this affects numerical functions such as `min` and `max`, and Boolean functions such as $=$ and `allDiff`. Normalisation is performed directly before the execution of any CSE algorithm in all the following experiments.

5.7. Experiment three: identical CSE

The simplest form of CSE that we consider is Identical CSE, which extracts sets of identical expressions. Suppose $x \times y$ occurs three times in a model. Identical CSE would introduce a new decision variable a and new constraint $x \times y = a$. The original occurrences of $x \times y$ would be replaced by a. In Savile Row, Identical CSE is implemented with Algorithm 3. Andrea Rendl's Tailor [31,59] and MiniZinc [65,39] also implement Identical CSE. In contrast to Tailor and MiniZinc, our algorithm is not tied to the process of flattening nested expressions into primitive expressions supported directly by the constraint solver. This is advantageous because it allows us to identify and exploit common subexpressions in expressions that do not need to be flattened.

The first step is to recursively traverse the model (by calling Algorithm 4) to collect sets of identical expressions. Algorithm 4 collects only expressions that are candidates for CSE. Atomic variables and constants are not candidates. Compound expressions are CSE candidates by default, with exceptions that depend on the target solver.

When the target is a SAT encoding we exclude all compound expressions that can be encoded as a single SAT literal. This avoids creating a redundant SAT variable that is equal to (or the negation of) another SAT variable, thus improving the

Algorithm 4 populateMap(*A*, *map*).

Require: *A*: Reference to an AST
Require: *map*: Hash table mapping expressions to lists
1: **if** *A* is a candidate for CSE **then**
2: Add *A* to list *map*[*A*]
3: **for all** *child* ∈ *A*.Children() **do**
4: populateMap(*child*, *map*)

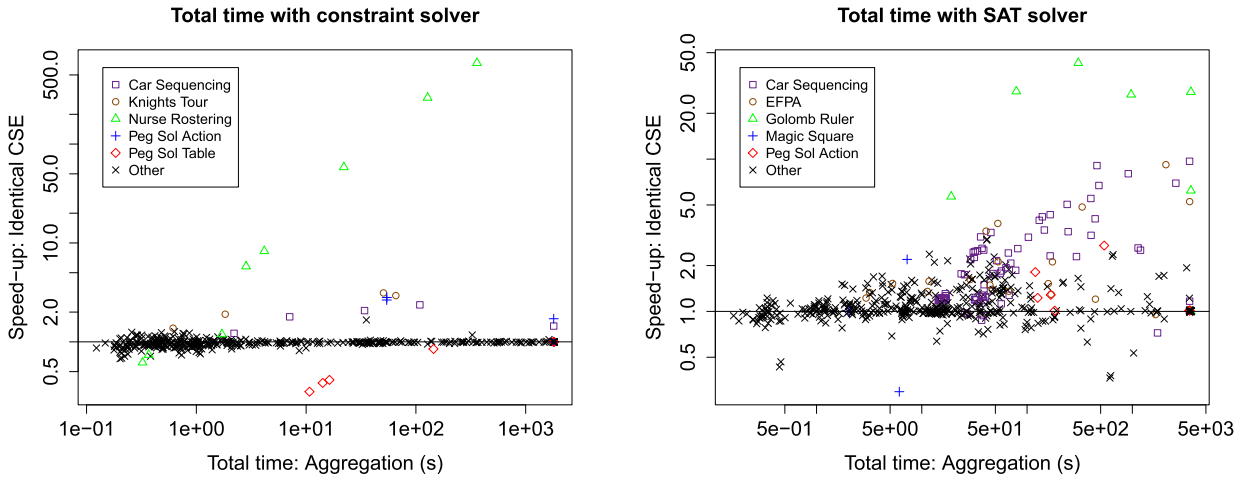

Fig. 6. Comparing IdentCSE to Aggreg (total time) with the CP solver (left) and the SAT solver (right). Axes are the same as Fig. 3.

encoding. The following expressions are not candidates: $x = c$, $x \neq c$, $x \leq c$, $x < c$, $x \geq c$, $x > c$, $\neg x$ (where x is a decision variable and c is a constant). When the target is Minion, $\neg x$ is not a candidate because negation is natively supported. Unary constraints $x = c$, $x \neq c$, $x \leq c$, $x < c$, $x \geq c$, $x > c$, $x \in \{c_1, c_2, \ldots\}$ (where x is a decision variable and c, c_1, c_2 are constants) are not candidates when they are contained in a conjunction or disjunction. When a unary constraint is propagated it is entirely represented in the domain so extracting unary constraints by CSE will not strengthen propagation. However, when a set of identical unary constraints must be extracted by general flattening (Section 3.4), Identical CSE will reduce the number of auxiliary variables and constraints. Minion natively supports conjunction and disjunction of any constraint [35] so there is no benefit from extracting unary constraints from conjunctions or disjunctions.

The second step of Identical CSE is to iterate through sets of expressions in decreasing size order (line 4). When an expression e is eliminated by CSE, the number of occurrences of any expressions contained in e is reduced. Therefore eliminating long expressions first may obviate the need to eliminate short expressions. For each set (of size greater than one) of identical expressions a new decision variable *aux* is created, and each of the expressions is replaced with *aux*. One of the expressions e in the set is used to create a new constraint $e = aux$. Crucially the new constraint contains the original object e so it is possible to extract further CSEs from within e.

The domain(e) function (line 9) obtains a finite domain for the new auxiliary variable. When extended domain filtering is off, domain(e) returns $\{\lfloor e \rfloor \ldots \lceil e \rceil\}$, otherwise it returns the filtered domain as described in Section 5.4. Finally, if an expression appears in the top-level conjunction, any other occurrences of it are replaced with `true`.

Identical CSE is a relatively straightforward optimisation and is implemented in a number of tools including MiniZinc [65] and IBM CP Optimizer Presolve [36].

Experimental results

Our experiments confirm that Identical CSE is beneficial for a number of problem classes. Fig. 6 plots our results comparing IdentCSE to Aggreg. When using the CP solver, Nurse Rostering exhibits the largest speed up with Identical CSE. There are identical sums shared between two groups of constraints, and extracting these strengthens propagation. For all other problem instances Identical CSE did not change the search tree. Excluding Nurse Rostering, the largest speed up is 3.13 for the Knights Tour size 7 instance, where expressions such as `|tour[1]/7 - tour[2]/7|` occur twice, and `tour[1]/7` occurs four times initially (twice after the containing expressions are extracted). Extracting these two types of common subexpression simply speeds up propagation. Peg Solitaire Table has no identical common subexpressions, however Identical CSE causes a substantial overhead on this class with Aggreg being (at most) 3.19 times faster than IdentCSE.[4]

[4] The overhead is not reproducible across different computers and JVM versions. In an informal test with one instance of Peg Solitaire Table using a different computer and JVM we found the overhead to be 1.16 times as opposed to 2.43 times in the experiment.

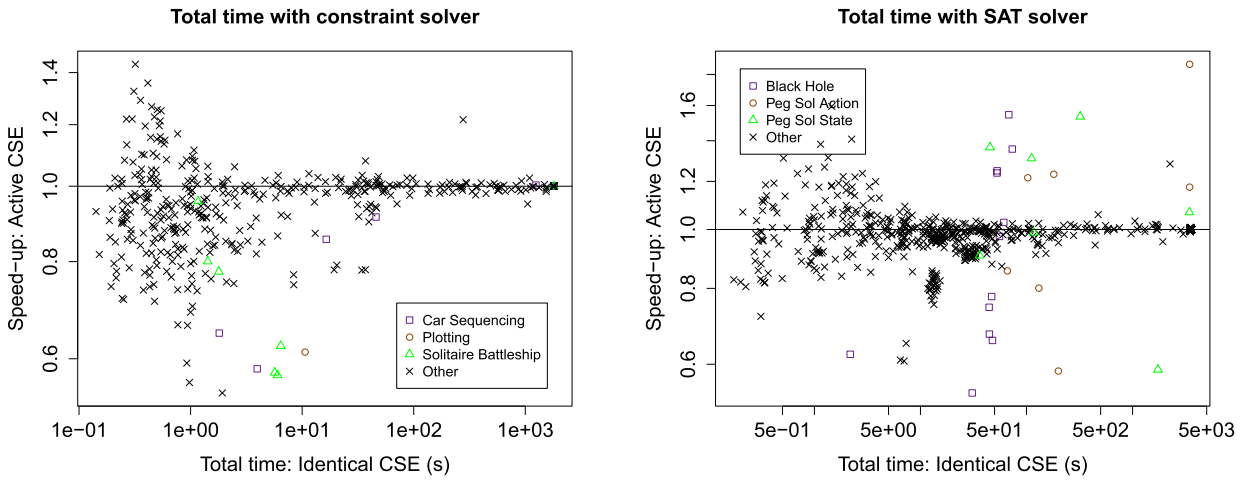

Fig. 7. Comparing ActiveCSE to IdentCSE (total time) with the CP solver (left) and the SAT solver (right). Axes are the same as Fig. 3.

With the SAT backend, Identical CSE is particularly helpful for Car Sequencing, EFPA and Golomb Ruler. In Car Sequencing the identical expressions are $x \in S$ for some variable x and integer set S that are contained in sums. For EFPA the decomposition of lexicographic ordering constraints (described in Section 3.5) is improved by Identical CSE and is also cross-connected with constraints requiring d differences between rows of the matrix. Golomb Ruler has identical common subexpressions within the decomposition of allDifferent, as described in the section above.

For the CP solver the geometric mean speed-up is 1.035 with confidence interval [0.988, 1.097]. The result is likely to be positive but does not quite reach the significance threshold. For the SAT solver the same statistic is 1.238 with confidence interval [1.190, 1.291]. Identical CSE will be switched on for all subsequent experiments.

5.8. Experiment four: active CSE

Active CSE extends Identical CSE by allowing non-identical expressions to be extracted using a single auxiliary variable. We reviewed the related work in Section 2.5. As an example, suppose we have $x = y$ and $x \neq y$ in the model. We can introduce a single Boolean variable a and a new constraint $a \leftrightarrow (x = y)$, then replace $x = y$ with a and $x \neq y$ with $\neg a$. For solvers that natively support negation (such as SAT solvers) $\neg a$ can be expressed in the solver input language with no further rewriting, so we have avoided expressing both $x = y$ and $x \neq y$ in the solver input language.

The Active CSE algorithm in Savile Row extends Algorithm 3. For each candidate expression e a simple transformation is applied to it (for example producing $\neg e$). Simplifiers and normalisation are applied to place $\neg e$ in the normal form. The algorithm then queries *map* to discover expressions matching the transformed expression.

Active CSE in Savile Row 1.6.5 applies four transformations: Boolean negation, arithmetic negation, multiply by 2, and multiply by -2. For example, $x - y + z$ matches $y - x - z$ using arithmetic negation. Rendl implemented Boolean negation active CSE in Tailor, along with active reformulations based upon De Morgan's laws and Horn clauses [59]. In Savile Row, the use of negation normal form obviates the use of the latter two. To our knowledge MiniZinc [65,39] does not implement Active CSE.

Experimental results

Fig. 7 plots our results comparing ActiveCSE to IdentCSE. With the CP solver Active CSE makes very little difference on any problem class. In most cases the search tree is unchanged, and the largest reduction in search nodes is 0.9%. When using the SAT solver, the instances that are affected by Active CSE are scattered, with some (in each affected class) improved and others degraded. The geometric mean speed-up quotient of total time is 0.936 when using the CP solver, with confidence interval [0.924, 0.949], and 0.976 when using the SAT solver with confidence interval [0.966, 0.985]. Active CSE will not be enabled for any subsequent experiment.

5.9. Experiment five: evaluation of X-CSE

We introduced the algorithm X-CSE in Section 4. In this experiment we look in detail at six problem classes where AC-CSs arise. We apply X-CSE to disjunction, conjunction, sum and product. These types are represented as a single AST node with n children in Savile Row. Two other AC types appear in our benchmarks: min and max (in one problem class each) and they exhibit no AC-CSs. X-CSE scales with the number of AC-CSs, so the cost of applying X-CSE to min and max would be minimal.

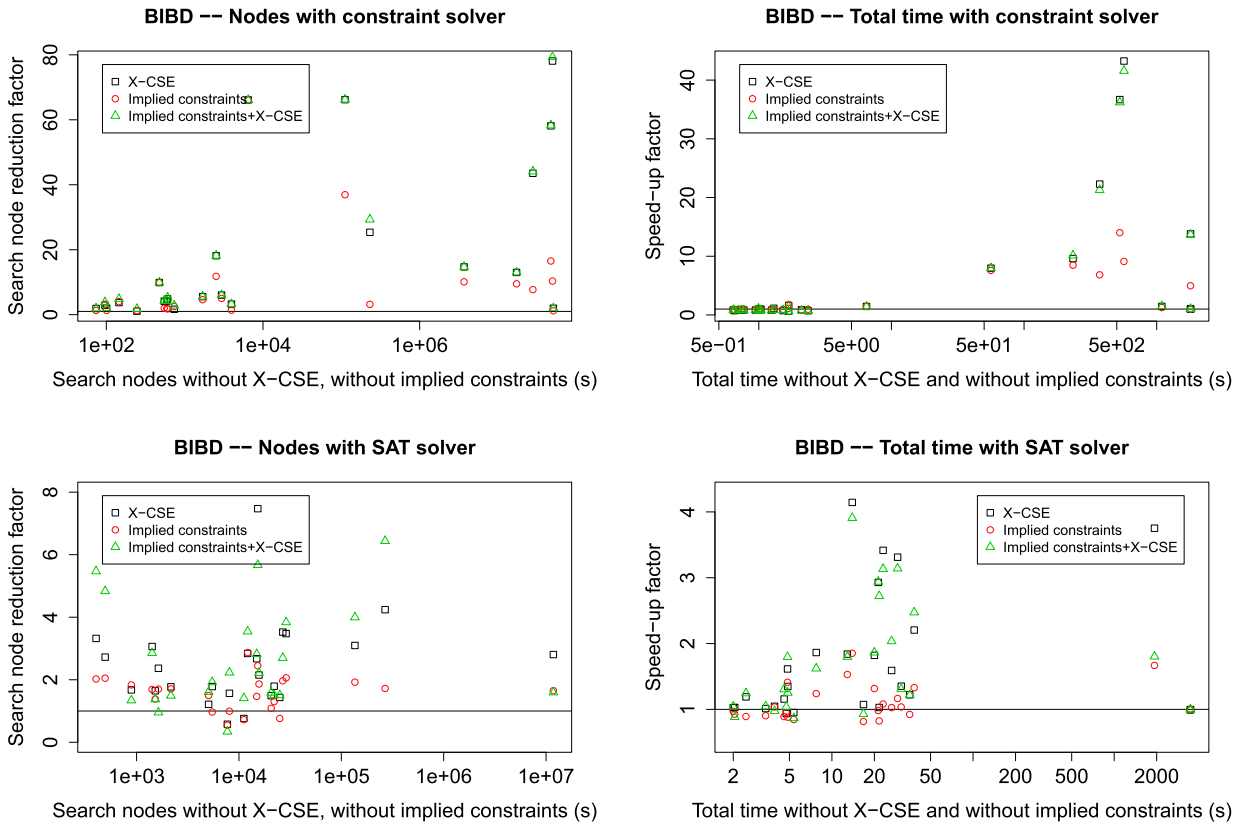

Fig. 8. Comparing X-CSE to IdentCSE. (Left) BIBD search nodes of instances that do not time out. (Right) BIBD total time. The upper plots are with the CP solver and the lower plots are with the SAT solver.

5.9.1. Case study 1: BIBD

The BIBD problem is described in high-level terms as our motivating example (Section 1.2). We use Meseguer and Torras' first model of BIBD [43], extended with Lex^2 symmetry breaking constraints [19]. BIBD is parameterised by (v, k, λ) and has constants $r = \frac{\lambda(v-1)}{k-1}$ and $b = \frac{\lambda v(v-1)}{k(k-1)}$. The model has a v by b matrix m of Boolean variables. Each of the v rows sums to r (row constraints), and each of the b columns sums to k (column constraints). The scalar product of each pair of rows has value λ:

$$\forall i_1, i_2 \in \{1 \dots v\} \,.\, i_1 < i_2 \rightarrow (\sum_{j=1}^{b} m[i_1, j] \times m[i_2, j]) = \lambda$$

Rows and columns are ordered lexicographically to remove some symmetry:

$$\forall i \in \{1 \dots v-1\} \,.\, m[i, \dots] \leq_{lex} m[i+1, \dots]$$
$$\forall j \in \{1 \dots b-1\} \,.\, m[\dots, j] \leq_{lex} m[\dots, j+1]$$

This model initially has no CSs of any type (identical, active or AC). Domain filtering (Section 5.4) assigns some of the variables (as seen in Fig. 1). As discussed in Section 1.2, the scalar product constraints are then simplified causing AC-CSs to appear among scalar product constraints, and between scalar product and row sum constraints.

We evaluated X-CSE on the 24 instances in Fig. 1 of Puget [55] and also with $(v, k, \lambda) = (7, 3, 20)$ and $(8, 4, 12)$. When using the CP solver, we found that 3 instances time out with IdentCSE and 2 of those also time out with X-CSE. For the remaining 23 instances where neither time out, X-CSE always decreases the node count. Fig. 8 (upper left) plots the reduction factor for the 23 instances. Harder instances tend to show a greater reduction in node count.

Fig. 8 (upper right) plots speed-up of total time with X-CSE. For the easiest instances, the reduction in node count does not cause a measurable difference in the run time of the constraint solver. The slight increase in total time is caused by the up-front cost of X-CSE. On the harder instances, Minion search takes up most of the total time and X-CSE speeds up search substantially by reducing the number of search nodes. Fig. 8 (upper right) peaks with instance $(12, 3, 4)$, which has a 78-fold reduction in nodes and speed up of 43.2 times. The time taken by Savile Row increased from 1.7 s to 2.6 s. X-CSE typically increases the number of constraints and auxiliary variables, reducing the node rate of the CP solver. With the SAT

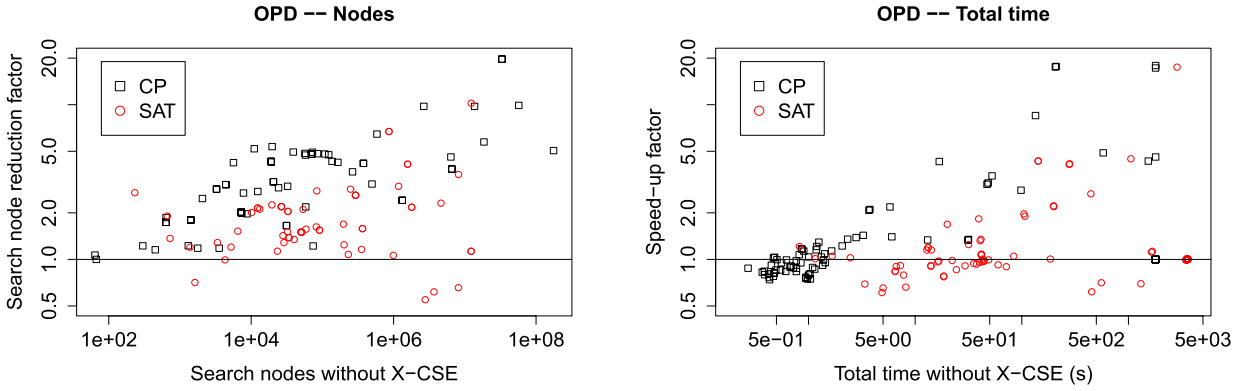

Fig. 9. Comparing X-CSE to IdentCSE. (Left) OPD search nodes of instances that do not time out. (Right) OPD total time.

solver we found that the same 2 instances timed out with both IdentCSE and X-CSE. For the other 24, X-CSE reduces the median search nodes for 22 instances and increases 2, as shown in Fig. 8 (lower left). The reduction in search nodes is much less dramatic than with the CP solver. Fig. 8 (lower right) plots speed-up of total time caused by X-CSE with the SAT solver. The peak speed-up quotient is 4.14 for instance $(8, 4, 12)$, and the median search nodes is reduced by 7.47 times.

5.9.2. Implied constraints for BIBD

Frisch, Jefferson, and Miguel [23] derived a set of implied constraints for BIBD that drastically improve the performance of the model. First they observed that the first two rows and first column of the BIBD can be assigned by manually reasoning about the constraints. Second, for each of the remaining rows i, the row sum constraint is split into two parts: all indices where row 1 is 0 sum to $r - \lambda$, and all other indices sum to λ (by combining the row sum to r with the scalar product constraint between rows 1 and i). The process is repeated with rows 2 and i, giving four constraints. These implied constraints are derived using an approach resembling manual AC-CSE.

The automated approach improves on Frisch et al. in two ways. First, domain filtering assigns not just the first two rows and first column but also parts of other rows and columns (see Fig. 1 for example). Second, X-CSE is able to link multiple scalar product constraints and a row constraint, whereas the implied constraints are each derived from one scalar product and one row constraint.

The implied constraints alone reduce node count with the CP solver (see Fig. 8 (upper left)) and for most instances with the SAT solver (lower left), but are not as effective as X-CSE. For most of the harder instances the implied constraints speed up solving but by a smaller degree than X-CSE. Adding the implied constraints then applying X-CSE is slightly more effective than X-CSE alone in reducing node count for the CP solver. However this does not translate to a clear improvement in search efficiency. Remarkably, X-CSE is able to improve the sophisticated model on most instances for both solvers.

5.9.3. Case study 2: optimal portfolio design

The Optimal Portfolio Design problem [20] (Problem 65 at www.csplib.org) is a challenging problem from computational finance. Following the description given in [20], A Collateralised Debt Obligation (CDO) is a portfolio of credit assets, such as bonds, loans, or credit default swaps. In order to manage risk, CDOs are divided into uniformly sized tranches each consisting of a subset of the underlying collection of credit assets. The task is to choose the constituents of the tranches so that each pair of tranches overlap as little as possible, in order to spread the risk.

The OPD problem can be seen as a more challenging optimisation variant of the BIBD problem, in which we are required to find v tranches each composed of a cardinality r subset of b credit assets, while minimising λ, the cardinality of the maximum intersection between any pair of tranches.

We evaluated X-CSE on a set of 101 OPD instances composed as follows: All 54 instances from Flener et al. [20]; a further 18 instances from CSPLib; and finally 29 further instances from the above BIBD experiments. The results are plotted in Fig. 9. Similarly to the BIBD results, when using the CP solver X-CSE produces a decrease in node count on all but one instance, and this reduction is more pronounced for the harder instances. In terms of total time, for the easiest instances there is again a small increase caused by the up-front cost of X-CSE, but on the harder instances where search is the dominant cost X-CSE speeds up the search substantially. Overall, for CP the average speed-up is 1.288 with confidence interval [1.107, 1.527]. When using the SAT solver, the average speed up is 1.217 with confidence interval [1.059, 1.428].

Although the structure of the OPD problem differs somewhat from that of the BIBD problem, in particular in that there is no sum on the columns of the design matrix, by a similar process to that described for BIBD, Savile Row is able to fix the first row of the OPD matrix and therefore identify sub expressions that X-CSE can exploit.

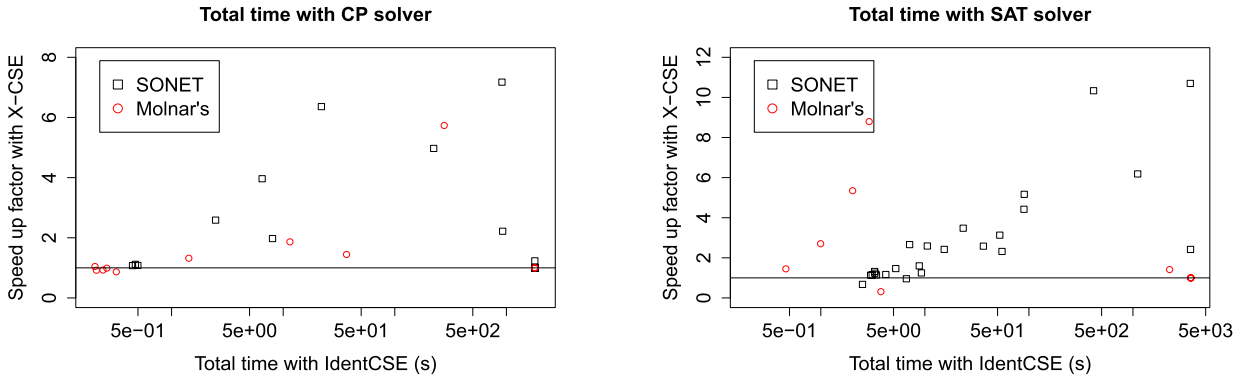

Fig. 10. Comparing X-CSE to IdentCSE for SONET and Molnar's Problem total time, with the CP solver (left) and the SAT solver (right).

5.9.4. Case study 3: the SONET problem

The SONET problem [64] is a network design problem where each node is connected to a set of *rings* (fibre-optic connections). The simplified SONET problem (Section 3 of [64]) has the following parameters: the number of nodes n, the upper limit on the number of rings m, the maximum number of nodes per ring r, and a set of pairs that must be connected. For each of these pairs there must exist a ring connected to both nodes. The number of node-ring connections is minimised.

The problem is modelled as follows. We have a Boolean matrix *rings* indexed by $[1 \ldots m, 1 \ldots n]$. $rings[a, b]$ indicates whether ring a is connected to node b. For each ring a we have the sum constraint $\sum_{b=1}^{n} rings[a, b] \leq r$. The connectedness constraint between two nodes b_1 and b_2 is expressed as a disjunction of sums:

$$\exists\, a \in \{1 \ldots m\}.\ (rings[a, b_1] + rings[a, b_2] \geq 2)$$

The minimisation function is the sum of *rings*. Rings are indistinguishable so we use lexicographic ordering constraints to order the rows of *rings*. The static variable ordering we use is the reading order of *rings* and value order is 0 then 1. This model is very simple and does not include implied or dominance constraints [64]. The constraints are already flat and only the minimisation sum needs to be flattened, thus only one auxiliary variable is created by Savile Row without X-CSE. There are AC-CSs among the connectedness constraints, the ring sum constraints and the minimisation sum.

We generated 24 instances with $n \in \{6 \ldots 13\}$, $r \in \{3, 4, 5\}$, and $m = 10$. The demand graph when $n = 13$ is Fig. 1 of Smith [64]. For smaller n we take the subgraph with vertices $\{(n+1) \ldots 13\}$, and edges adjacent to these vertices, removed. Fig. 10 (left) plots the speed-up factor for X-CSE when using the CP solver. For IdentCSE with the CP solver, all instances with $n \geq 10$, also $n = 9, r = 4$ and $n = 9, r = 3$ time out. X-CSE is able to solve to optimality one additional instance $n = 10, r = 5$, which solved just within the time limit and appears on the right edge of the plot with a speed-up of 1.23. X-CSE improves solving speed for all but the most trivial instances, with a peak speed-up of 7.17. When using the SAT solver as shown in Fig. 10 (right), IdentCSE solves all but two instances of SONET to optimality, and all instances are solved to optimality with X-CSE. The two instances that are not solved with IdentCSE appear on the far right of the plot. Instance $n = 13, r = 5$ exhibits the largest speed-up of 10.7 times, and it is one of the two that timed out with IdentCSE.

5.9.5. Case study 4: Molnar's problem

Molnar's problem [22] (CSPLib problem 035 [21]) is to find a square matrix M of integers. The model has two parameters: the size k (i.e. M has size $k \times k$) and the maximum absolute value of integers in M, named d. The initial domain of each element of M is $\{-d \ldots -2\} \cup \{0\} \cup \{2 \ldots d\}$. The first constraint is that the determinant of M is 1 or -1 (following the model of Frisch et al. [22]). For the second constraint we construct another matrix S where each entry of S is the square of the corresponding entry of M. The determinant of S must also be 1 or -1. We used the Leibniz formula for determinants, and expressed a^2 as $a \times a$ to allow more AC-CSs of products. When $k = 3$ we have the following two matrices and two constraints. In addition we break symmetry on M by lexicographically ordering rows and columns.

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, S = \begin{bmatrix} a^2 & b^2 & c^2 \\ d^2 & e^2 & f^2 \\ g^2 & h^2 & i^2 \end{bmatrix}$$

$$|M| = aei - afh + bfg - bdi + cdh - ceg \in \{-1, 1\}$$

$$|S| = aaeeii - aaffhh + bbffgg - bbddii + ccddhh - cceegg \in \{-1, 1\}$$

There are multiple AC-CSs of products, for example aa and aei. Some connect the two sums, and others connect terms within one sum. X-CSE is able to extract a particular AC-CS from the same product more than once on this problem. Consider aei: extracting it once creates a new constraint $aei = x$ and modifies the existing expression aei to x, and the expression $aaeeii$ to $x \times aei$. Now X-CSE extracts aei a second time from the new constraint and one of the modified expressions, creating a second auxiliary variable (that will later be unified with x).

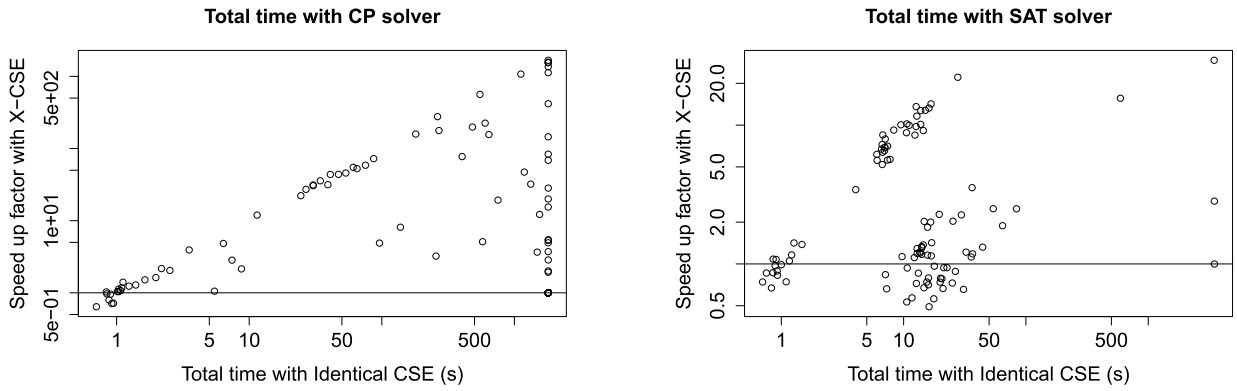

Fig. 11. Comparing X-CSE to IdentCSE for Killer Sudoku 16×16 total time, with the CP solver (left) and the SAT solver (right).

Fig. 10 (left) plots the speed-up quotient with the CP solver for Molnar's Problem on the twelve instances where $k \in \{2 \ldots 4\}$ and $d \in \{2 \ldots 5\}$. X-CSE is more useful for the more difficult instances. Instances where $k = 4$ and $d \geq 3$ time out with both IdentCSE and X-CSE. The peak speed-up quotient is 5.73. Fig. 10 (right) plots the speed-up quotient with the SAT solver. Both IdentCSE and X-CSE time out when $k = 4$ and when $k = 3$ and $d \geq 4$. The peak speed-up quotient is 8.79.

5.9.6. Case study 5: Killer Sudoku

The standard Killer Sudoku [45] has a 9×9 grid where each row and column are allDifferent, and the nine non-overlapping 3×3 subsquares are also allDifferent. Each slot in the grid is initially empty and takes a digit $1 \ldots 9$. Clues are sets of squares that sum to a given value (and are also allDifferent). We found 9×9 Killer Sudoku instances to be very easy, so we used a 16×16 grid with 16 4×4 subsquares, and each slot takes a number $1 \ldots 16$. We generated 100 instances by first creating 100 random assignments to the 16×16 grid. These assignments satisfy the row, column and subsquare allDifferent constraints. Second, for each random assignment we generated clues of lengths 1 to 5 using random walks from random starting points. (For brevity we omit full details of instance generation: Instances are included with Savile Row 1.6.5.) Traditional Killer Sudoku puzzles have exactly one solution. The random 16×16 instances may be unsatisfiable and may have multiple solutions.

AC-CSE alone does nothing because the sums in the clues are the only AC expressions and they do not overlap. However the sums overlap with allDifferent constraints. Each allDifferent constraint on a row, column or subsquare represents a permutation of $\{1 \ldots 16\}$ which sum to 136. Savile Row automatically adds implied sum constraints as described in Section 4.4. In the first tailoring process the implied sums will be $\sum X = 136$ where X is the set of variables in the row, column or subsquare. In the second tailoring process some variables may be assigned and deleted, with the assigned value removed from other variables in the same row, column or subsquare. The implied sums for rows, columns and subsquares will be of the form $\sum X = c$ but the constant c may differ. X-CSE then finds AC-CSs among rows, columns, subsquares and clues.

Fig. 11 (left) plots the speed-up quotient for Killer Sudoku when using the CP solver. Without X-CSE, 44 instances timed out. With X-CSE, 24 instances timed out. As the instances become more difficult, the trend is towards greater speed-up by X-CSE. The plot peaks at 1671 times faster. On this instance, with the IdentCSE configuration Savile Row took 1.15 s and Minion timed out after exploring 9,188,782 nodes. With X-CSE, Savile Row took 0.95 s and Minion took 0.13 s to explore 4 nodes. The geometric mean speed-up quotient of total time is 9.72 over 100 instances.

When using the SAT solver the results are still very positive, but not as striking. Fig. 11 (right) shows that X-CSE improves most instances but degrades performance on others. The geometric mean speed-up quotient of total time is 2.119 with confidence interval [1.720, 2.625]. IdentCSE times out for three instances and X-CSE is able to solve two more. There are two main groups of instances, one where X-CSE enables domain filtering to solve or almost solve the instances (making the SAT encoding very small), and another where X-CSE is (for the most part) increasing the size of the SAT encoding. In 91 of 100 instances, if the SAT encoding has more variables then the total time is greater.

5.9.7. Case study 6: car sequencing

The Car Sequencing problem [51] (CSPLib problem 1) serves as a prototypical example of a sequencing problem, of which many other examples exist, and shows how common subexpressions can arise and be exploited. The problem is an abstracted factory scheduling problem, and is NP-hard on its own [18]. The goal is to manufacture cars on an assembly line. The cars are not identical: there are a set of *options* that may be included (for example, air conditioning). Each option is fitted to the cars at a station on the assembly line, and each station s has a capacity constraint: it can take at most p_s of every q_s consecutive cars, where p_s and q_s are given. For each class of (identical) cars, the number of that class to be manufactured is given along with the set of options required by that class.

We represent each car class with an integer and the sequence of cars as a sequence of integer variables *seq*. A global cardinality constraint [56] ensures the required number of each class is manufactured. The station capacity constraints are

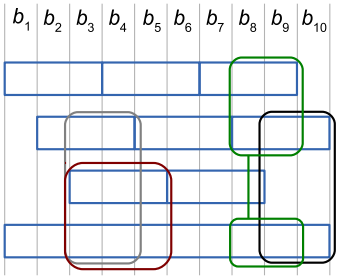

Fig. 12. Car Sequencing AC-CSE example, with 10 cars in total. Blue boxes represent scopes of sums over the Boolean variables $b_1 \ldots b_{10}$. The first three rows represent station capacity constraints (each one a sum ≤ 2) and the last row represents the implied constraint (sum $= 5$). Curved boxes represent AC-CSs. The pair on the left (grey and red) can both be extracted, and the pair on the right (black and green) conflict and cannot both be extracted by X-CSE. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

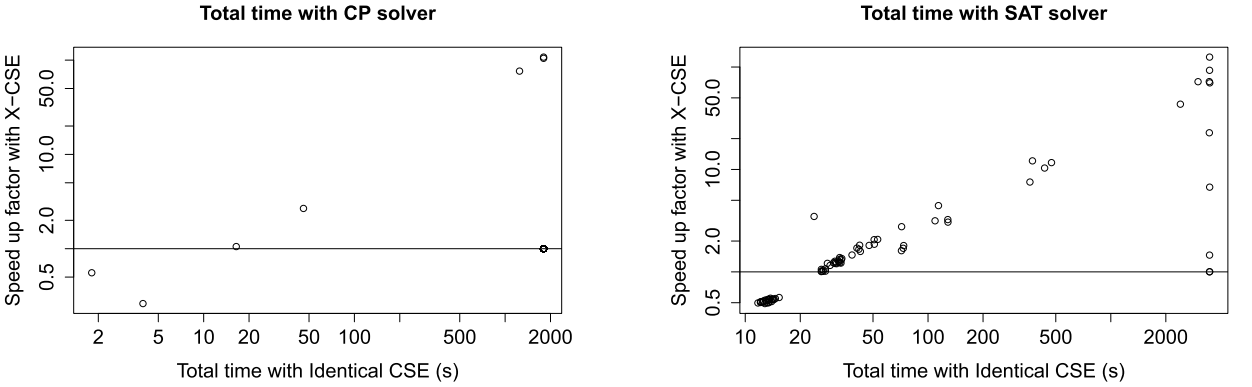

Fig. 13. Comparing X-CSE to IdentCSE for Car Sequencing total time, with the CP solver (left) and the SAT solver (right).

expressed as sums $\leq p_s$ of expressions $(seq[i] \in S_{opt})$ where S_{opt} is the set of car classes that require *opt*. Finally an implied constraint is added for each option that states the option is fitted the required number of times over the whole sequence. These constraints are also sums of $(seq[i] \in S_{opt})$ expressions. The model is much simpler than others in the literature [58, 33,11,46] and as it stands it will not propagate well in a conventional CP solver. The search branches on *seq* in order and assigns values in ascending order.

The expressions $x \in S_{opt}$ are extracted by Identical CSE leaving sums of Boolean variables. For each option, the station capacity constraints overlap with each other and also with the implied constraint for the option. X-CSE connects together all the constraints related to a single option, but is not able to exploit every AC-CS since some pairs conflict. Fig. 12 shows a conflicting pair of AC-CSs on a small example.

Fig. 13 plots our results with the CP solver and the SAT solver. The SAT solver is more successful in solving these instances both with IdentCSE and X-CSE. For both solvers X-CSE improves the model considerably with peak speed-up quotients of over 100 times on the most difficult instances solvable within the time limit.

It is likely that an aggregation approach (producing a set of global sequence constraints [58] or a pre-defined encoding [33,11]) would perform better than X-CSE on this problem class, however aggregation would also be less general than X-CSE.

Finally, applying X-CSE generates part of the Partial Sums encoding of sequence by Brand et al. [11]. Partial Sums introduces a variable for each contiguous subsequence of length 2 to q_s (for a station s), which is a superset of the variables X-CSE introduces. Enforcing bound consistency on Partial Sums guarantees GAC on the sequence constraint, whereas X-CSE has no similar guarantee.

While we have experimented only on car sequencing, we expect the pattern of overlapping sums seen in Fig. 12 to arise in other sequencing problems, and therefore for AC-CSE to be widely applicable in sequencing problems.

5.9.8. Summary plots for X-CSE

Fig. 14 compares X-CSE to IdentCSE for all problem classes. The geometric mean speed-up is 1.492 when using the CP solver with confidence interval [1.326, 1.693], and 1.323 when using the SAT solver with confidence interval [1.236, 1.422]. There are two notable problem classes (other than the case studies above): Peg Solitaire Action (with both solvers), and Graph Colouring (with the SAT solver only). Applying X-CSE to Peg Solitaire Action adds a considerable overhead to Savile Row (owing to a large number of long conjunctions nested within constraints) and when targeting the CP solver does not

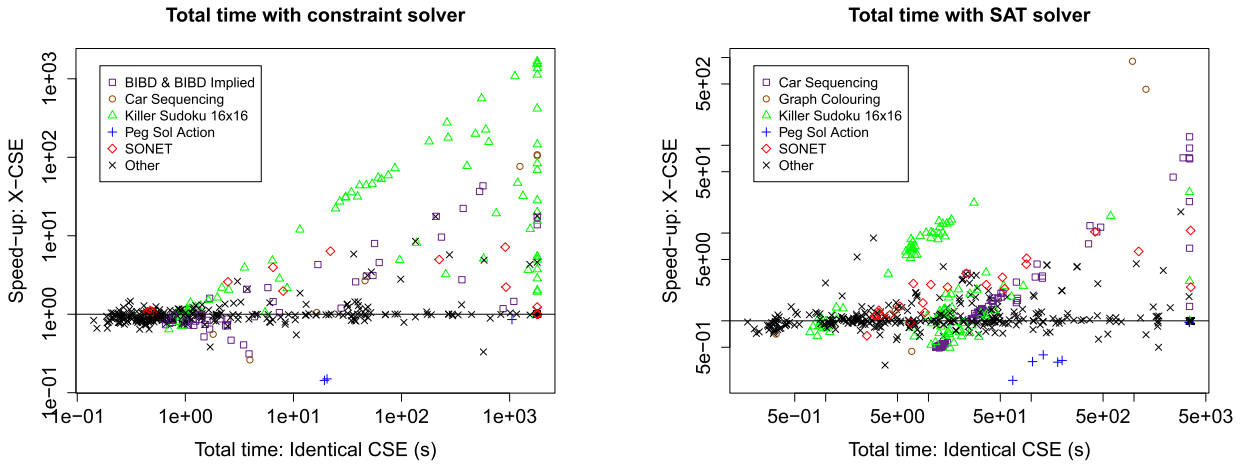

Fig. 14. Comparing X-CSE to IdentCSE (total time) with the CP solver (left) and the SAT solver (right).

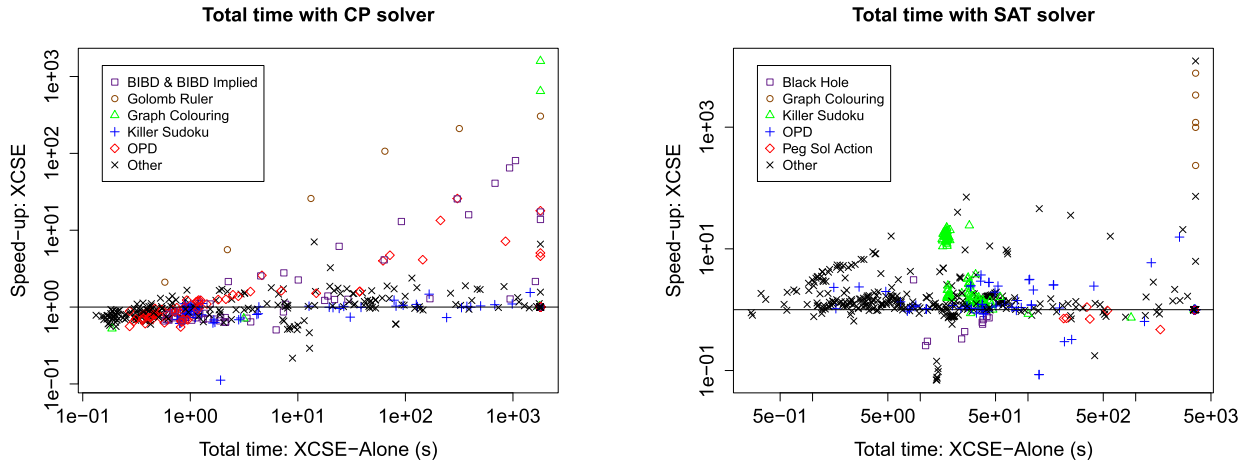

Fig. 15. Comparing X-CSE with other reformulations to X-CSE alone (total time) with the CP solver (left) and the SAT solver (right).

change the search tree. For SAT, more variables are generated, the number of search decisions is increased and Lingeling takes longer.

In Graph Colouring with the SAT solver, the two instances that exhibit a very large speed-up are unsatisfiable because of a limit on the number of colours. In both instances, some of the cliques of disequalities are larger than the number of colours. The cliques are collected to an AllDifferent with more variables than domain values. In this case the pass that generates implied sums from AllDifferent creates a `false` constraint so the SAT encoding becomes trivial.

5.10. Interaction of X-CSE with other reformulations

We have shown that X-CSE is a valuable addition to the set of reformulations we have accumulated so far: Identical CSE, aggregation, standard domain filtering, and variable unification. However, the interaction of X-CSE with the other reformulations is not clear. Here we compare X-CSE alone (referred to as X-CSE-Alone) to X-CSE with all the other reformulations accumulated so far (named simply X-CSE in Table 1).

Fig. 15 plots X-CSE against X-CSE-Alone. With the CP solver many problem classes perform better with X-CSE than X-CSE-Alone. BIBD and OPD both have no AC-CSs in the initial model but domain filtering, variable unification and simplifiers create a set of AC-CSs that are then extracted by the X-CSE algorithm. For both Golomb Ruler and Graph Colouring aggregation is essential. In common with experiment one (Section 5.4) there are many easy instances that are solved in less than 10 seconds with X-CSE-Alone, and for most of these instances the overhead of the full set of reformulations is not repaid. The geometric mean speed-up of the full set of instances is 1.060 with confidence interval [0.990, 1.142]. If we exclude instances where X-CSE-Alone takes less than 10 seconds total time, the geometric mean speed-up is 1.795 with confidence interval [1.476, 2.232].

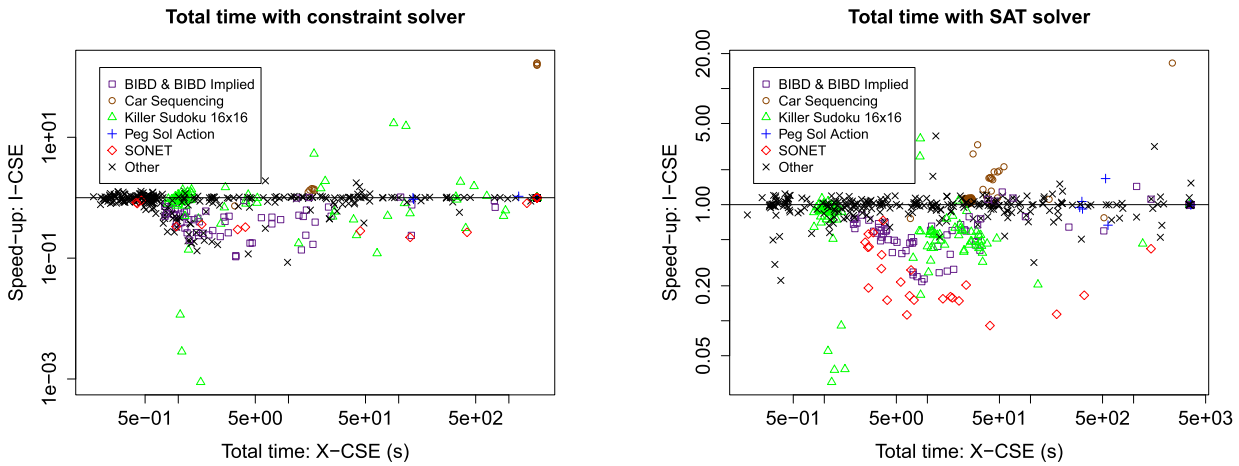

Fig. 16. Comparing I-CSE to X-CSE (total time) with the CP solver (left) and the SAT solver (right).

With the SAT solver X-CSE is clearly better than X-CSE-Alone with a geometric mean speed-up of 1.853 with confidence interval [1.708, 2.014]. If we compare IdentCSE (i.e. Identical CSE, aggregation, standard domain filtering and variable unification) to Simp the geometric mean speed-up is 1.626 with confidence interval [1.531, 1.730]. The difference in average speed-up indicates that there must be a significant interaction between X-CSE and the four other reformulations. If we examine individual problem classes the results are spread. For example OPD instances are all changed in a similar way by X-CSE (compared to X-CSE-Alone), and most are improved by X-CSE but there are some difficult OPDs where X-CSE-Alone is faster.

5.11. Experiment six: I-CSE, I-CSE-NC and I-CSE-Iter

We now compare X-CSE to the alternative AC-CSE algorithm I-CSE and its variants (described in Sections 2.6 and 4.3). In each of the I-CSE configurations, the I-CSE algorithm is applied to sums and products (as in the original paper by Araya et al. [2]) and the X-CSE algorithm is applied to conjunctions and disjunctions. Therefore when comparing I-CSE with X-CSE, we are comparing their performance on sums and products only, while treating conjunctions and disjunctions identically.

Experimental results

Fig. 16 plots the speed-up factor for I-CSE compared to X-CSE. With the CP solver, I-CSE performs better on a very small number of instances, notably four instances of Car Sequencing that can be solved with I-CSE and not with X-CSE. These four instances exhibit a speed-up of 155 to 169 times. As discussed in Section 5.9.7, Car Sequencing has a large number of conflicting pairs of AC-CSs that I-CSE can exploit. Two Killer Sudoku instances time out with I-CSE and are solved with X-CSE. X-CSE can exploit AC-CSs among a subsquare, a row or column, and a clue that are not available to I-CSE because I-CSE considers expressions pairwise. Overall X-CSE is preferable for SONET, Killer Sudoku, and BIBD. When using the SAT solver, broadly the same problem classes are improved or deteriorated when comparing I-CSE to X-CSE.

When using the CP solver the geometric mean speed-up is 0.737 (confidence interval [0.689, 0.786]) and it is 0.797 (confidence interval [0.758, 0.837]) when using the SAT solver. With the CP solver both algorithms time out on 174 instances, albeit different sets of 174 instances. With the SAT solver I-CSE times out for 68 instances, and X-CSE times out for 67. Overall it seems that X-CSE strikes a better balance between increasing the size of the constraint set and strengthening propagation.

In previous work [47] we found that I-CSE often performed better than X-CSE on Killer Sudoku, whereas here only a small number of Killer Sudoku instances are improved by I-CSE compared to X-CSE. The context here is different in a number of ways. For example, we have added aggregation that can lead to additional implied sum constraints on this problem class.

I-CSE-NC is a lighter weight version of I-CSE that does not copy expressions, so it cannot exploit both of a conflicting pair of AC-CSs. On average it is somewhat worse than I-CSE, as discussed in [48, Section D]. I-CSE-Iter extends I-CSE by iterating until a fixpoint is reached. On average it very slightly improves the performance of the CP solver and very slightly degrades the SAT solver when compared to I-CSE. Full results are given elsewhere [48, Section D].

5.12. Experiment seven: active associative–commutative CSE

Standard AC-CSE requires the extracted subexpressions to be syntactically identical apart from reordering. Normalisation helps by reducing semantic identity to syntactic identity in some cases, but we can extend the reach of AC-CSE further by

considering simple transformations as in Active CSE. If we consider some transformation t (for example, multiplying by two then restoring the normal form), an Active AC-CS e is a subexpression that is contained in one set of AC expressions as-is, and the transformed version $t(e)$ is contained in another set of AC expressions. Where e occurs, it is replaced with a new auxiliary variable a, and $t(e)$ is replaced with $t(a)$.

An example is found in the reservoir management problem described by Choi and Lee [14]. Their first model has differences $x_i - y_i$ in one set of sums and $y_i - x_i$ in another, where i is the day from a period of several months. This is improved by introducing a new set of variables for the differences $x_i - y_i$. Their improvement, made manually, is equivalent to Active AC-CSE with an arithmetic negation transformation.

A transformation t used in Active AC-CSE may change the type of the AC expression (i.e. e and $t(e)$ are different types). For example, consider a transformation $t(e)$ that takes the Boolean negation of e then restores the normal form. If e is a disjunction, $t(e)$ will be a conjunction and vice versa, so an algorithm that implements Active AC-CSE for this transformation must track both conjunctions and disjunctions.

In this paper we investigate one case of Active AC-CSE and propose an algorithm that is an extension of X-CSE. The case we consider is arithmetic negation of sums. The transformation takes a sum $c_1e_1 + \cdots + c_ne_n$ (where c_i are integers and e_i may be any expression) and simply negates each one of the coefficients c_i. This captures one case where the coefficients are proportional (as in rBC2-Y and rBCall-Y [9]).

The Active AC-CSE algorithm (named Active X-CSE) is an extension of Algorithm 1. The first step, populating *map*, remains unchanged. The heuristic (called on line 5) is adapted to select a pair where the pair and its negation together have the most occurrences. Once a pair has been selected, the lists for the pair and its negation are retrieved from *map* and the two lists are filtered (as on line 8). If the total length of the two lists is greater than one, then an (active or standard) AC-CS is extracted. The process is similar to lines 10–20 of Algorithm 1. The largest possible common set is extracted from the expressions in the two lists and replaced in each case with a reference to a new auxiliary variable *aux* or $-aux$.

Finally, the algorithm is optimised (similarly to X-CSE) by using a second hash table recording the global number of occurrences of every expression contained in a sum. If an expression e_1 occurs only once, then it cannot take part in an AC-CS and no pairs $\{e_1, e_2\}$ are stored in *map*. However (unlike X-CSE) the coefficient is not included as part of the expression when counting occurrences.

In this experiment we apply Active X-CSE to sums and X-CSE to conjunctions, disjunctions and products. In all other respects the Active X-CSE configuration is identical to the X-CSE configuration. We found that the benchmark set does contain some Active AC-CSs, however there were no substantial improvements to solving performance. Full results are given elsewhere [48, Section E].

5.13. Experimental conclusions

In the experiments above we have recommended the following set of reformulations: variable unification, standard domain filtering, aggregation, Identical CSE and X-CSE. To gauge how effective this set of recommended reformulations are, Fig. 17 compares using all of them to Savile Row with simplifiers only. Considering only time taken by the solvers, both CP and SAT models are massively improved: the geometric mean speedup is 3.627 for the CP solver and 6.614 for SAT. This is achieved entirely automatically, where the comparison is not with a straw man but a model written by hand and tailored by Savile Row in baseline form. The overhead of extra reformulation reduces the overall speedup. For CP, the geometric mean speed-up of total time is 1.568 with confidence interval [1.359, 1.823] and 32 more instances solved. For SAT it is 2.147 with confidence interval [1.973, 2.340] and 16 more instances solved. For the CP solver there is a general trend that harder instances benefit more from reformulation, and for some problem classes the gains can be hundreds or thousands of times. For instances that Simp solves in more than 10 seconds, the geometric mean speedup of total time is 5.959. We conclude that the recommended reformulations are well worthwhile, especially on harder instances where solver time dominates.

The largest gains we have seen have come from AC-CSE, across a diverse range of problem classes including a factory planning problem, a puzzle, combinatorial designs, and a network design problem. For the most part both the CP and SAT solvers have benefited from AC-CSE, implemented with the X-CSE algorithm (including the generation of implied sum constraints from global constraints). Depending on the problem class, the peak speed-up may be tens, hundreds or even thousands of times. In most cases AC-CSE is improving the formulation of overlapping sum constraints.

Identical CSE is also beneficial over a wide range of problem classes, particularly with the SAT solver where it avoids the overhead of multiple encodings of the same expression. Aggregation is also important when it applies, and very low-cost when it does not. Other reformulations have proven to be important preparatory steps that complement CSE (particularly AC-CSE): variable unification, domain filtering and applying simplifiers all have a role in revealing common subexpressions. Alone, variable unification and domain filtering provide a small benefit.

Both Identical CSE and AC-CSE were extended with active transformations to match non-identical but related terms (Active CSE and Active X-CSE). Neither of these extensions yielded a significant improvement. Our benchmark models were mainly written by expert users. It would be interesting to see whether novice users would write messier models (perhaps expressing the same concept in multiple ways) that would be amenable to Active CSE or Active X-CSE.

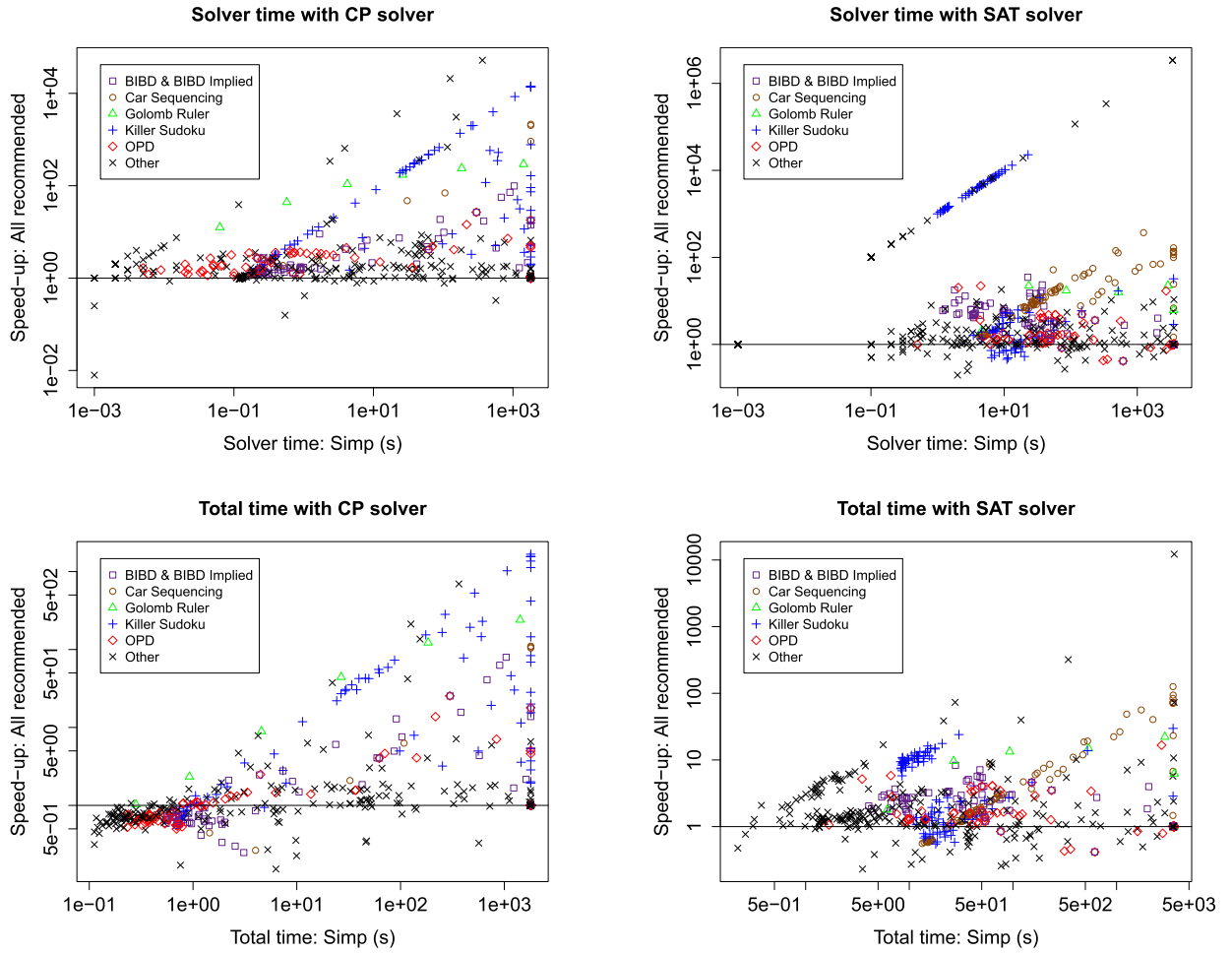

Fig. 17. Comparing X-CSE to Simp with the CP solver (left) and the SAT solver (right). The upper plots show solver time only and the lower plots show total time.

6. Conclusions and future work

We have investigated a range of reformulations, described algorithms to perform them, and implemented them in the system Savile Row. Some of these reformulations are novel improvements on the existing work. Furthermore, we have discovered sequences of reformulations that are much more effective than any single reformulation.

Each reformulation was evaluated on a set of 50 problem classes with 596 instances in total, in a very extensive set of experiments where reformulations were switched on one by one and their effects analysed as part of the system. Throughout we reported the cost of performing a reformulation, including on instances where the model was not changed. Two quite different target solvers were used in the experiments: Minion, a backtracking CP solver, and Lingeling, a state-of-the-art SAT solver with conflict learning and inprocessing. Most of the reformulations prove to be effective for both target solvers, which is encouraging for the study of automated modelling techniques.

We have investigated four different forms of common sub-expression elimination (CSE) that are able to exploit cases where equivalent or similar expressions appear in different constraints. We have introduced and described a new algorithm, X-CSE, to perform Associative–Commutative CSE (AC-CSE) as an automated modelling step for finite domain constraint satisfaction problems. X-CSE is able to find common subexpressions which are particularly effective in six benchmark problems: BIBD, OPD, SONET, Killer Sudoku, Car Sequencing and Molnar's Problem. Of particular importance, X-CSE interacts with other reformulations, thereby magnifying the power of those techniques and X-CSE. We suggest that X-CSE is preferable to an earlier algorithm for AC-CSE, namely I-CSE, because it is able to exploit frequently occurring short common subexpressions. In our experiments X-CSE outperformed I-CSE in most cases. We conclude that X-CSE is a valuable addition to the armoury of automated constraint modelling techniques, both alone and in combination with other techniques.

We found that a wide range of reformulations are valuable, including variable unification, domain filtering and aggregation. The most dramatic gains came from the X-CSE algorithm for AC-CSE, often when combined with the generation of

implied sum constraints, variable unification, domain filtering and simplifiers. With this combination, we saw peak speed increases of hundreds and even thousands of times.

The success of the CSE methods immediately suggests two avenues of future work. CSE improves the formulation when two expressions overlap by more than one variable, and the types of the two expressions are appropriate. For Identical and AC-CSE the types must be the same. Our first avenue of future work is to examine every pair of types in the language for opportunities to exploit overlapping expressions. Distinct types would necessarily require a variety of approaches to exploit the overlap. In some cases a new variable may help to transfer information between the two expressions, and in others a new constraint could capture the interaction. Some pairs of overlapping constraints are intractable (such as two allDifferent constraints), in which case we would investigate approximations.

The second avenue of future work is to improve the generation of implied constraints that feed into AC-CSE. At present, only sums are created, all coefficients are 1, only allDifferent and GCC are allowed as source constraints, and the scope of the implied sum constraint is the scope of the source constraint. Firstly, when a sum overlaps with a subset of the variables in a source constraint, tighter bounds may be obtained by generating the implied sums on the overlap variables only. Secondly, implied sums can be generated with coefficients that match other sums in the model, enabling AC-CSE. Thirdly, other AC expressions such as products could be generated when appropriate. Finally, a generic algorithm could be devised to generate implied sums from any constraint with appropriate properties, for example where all variables are interchangeable.

Acknowledgements

We would like to thank the EPSRC for funding this work through grants EP/H004092/1, EP/K015745/1, EP/M003728/1, and EP/P015638/1. In addition, Dr Jefferson is funded by a Royal Society University Research Fellowship.

References

[1] T. Achterberg, R.E. Bixby, Z. Gu, E. Rothberg, D. Weninger, Presolve Reductions in Mixed Integer Programming, Tech. rep., Zuse Institute, Berlin, 2016.
[2] I. Araya, B. Neveu, G. Trombettoni, Exploiting common subexpressions in numerical CSPs, in: Principles and Practice of Constraint Programming, CP 2008, 2008, pp. 342–357.
[3] M.G. de la Banda, K. Marriott, R. Rafeh, M. Wallace, The modelling language Zinc, in: Proc. 12th International Conference on the Principles and Practice of Constraint Programming, CP 2006, 2006, pp. 700–705.
[4] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, CVC4, in: Proceedings of Computer Aided Verification, 2011, pp. 171–177.
[5] N. Beldiceanu, H. Simonis, A constraint seeker: finding and ranking global constraints from examples, in: Proceedings of the 17th International Conference on the Principles and Practice of Constraint Programming, CP 2011, 2011, pp. 12–26.
[6] D. Bergman, J.N. Hooker, Graph coloring inequalities from all-different systems, Constraints 19 (4) (2014) 404–433.
[7] C. Bessière, S. Cardon, R. Debruyne, C. Lecoutre, Efficient algorithms for singleton arc consistency, Constraints 16 (1) (2011) 25–53.
[8] C. Bessière, R. Coletta, T. Petit, Learning implied global constraints, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007, 2007, pp. 44–49.
[9] C. Bessière, A. Paparrizou, K. Stergiou, Strong bounds consistencies and their application to linear constraints, in: Proceedings of the 29th International Conference on Artificial Intelligence, 2015, pp. 3717–3723, AAAI-15.
[10] A. Biere, M. Heule, H. van Maaren, Handbook of Satisfiability, vol. 185, IOS Press, 2009.
[11] S. Brand, N. Narodytska, C.G. Quimper, P. Stuckey, T. Walsh, Encodings of the sequence constraint, in: Proc. 13th Principles and Practice of Constraint Programming, CP 2007, 2007, pp. 210–224.
[12] J. Charnley, S. Colton, I. Miguel, Automatic generation of implied constraints, in: Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006, 2006, pp. 73–77.
[13] C.W. Choi, W. Harvey, J.H.M. Lee, P.J. Stuckey, Finite domain bounds consistency revisited, in: Proc. 19th Australian Joint Conference on Artificial Intelligence, AI 2006, 2006, pp. 49–58.
[14] C.W. Choi, J.H.M. Lee, Solving the salinity control problem in a potable water system, in: Proc. 13th Principles and Practice of Constraint Programming, CP 2007, 2007, pp. 33–48.
[15] J. Cocke, Global common subexpression elimination, ACM SIGPLAN Not. 5 (7) (1970) 20–24.
[16] G.J. Duck, L. De Koninck, P.J. Stuckey, Cadmium: an implementation of ACD term rewriting, in: Proceedings of the International Conference on Logic Programming, ICLP 2008, 2008, pp. 531–545.
[17] N. Eén, A. Biere, Effective preprocessing in SAT through variable and clause elimination, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2005, pp. 61–75.
[18] B. Estellon, F. Gardi, Car sequencing is NP-hard: a short proof, J. Oper. Res. Soc. 64 (10) (2013) 1503–1504.
[19] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh, Breaking row and column symmetries in matrix models, in: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming, CP 2002, 2002, pp. 462–476.
[20] P. Flener, J. Pearson, L.G. Reyna, O. Sivertsson, Design of financial CDO squared transactions using constraint programming, Constraints 12 (2007) 179–205.
[21] A. Frisch, C. Jefferson, I. Miguel, CSPLib problem 035: Molnar's problem, http://www.csplib.org/Problems/prob035.
[22] A.M. Frisch, C. Jefferson, I. Miguel, Constraints for breaking more row and column symmetries, in: Proceedings CP 2003, 2003, pp. 318–332.
[23] A.M. Frisch, C. Jefferson, I. Miguel, Symmetry-breaking as a prelude to implied constraints: a constraint modelling pattern, in: Proc. 16th European Conference on Artificial Intelligence, ECAI 2004, 2004.
[24] A.M. Frisch, I. Miguel, T. Walsh, CGRASS: a system for transforming constraint satisfaction problems, in: B. O'Sullivan (Ed.), International Workshop on Constraint Solving and Constraint Logic Programming, in: Lect. Notes Comput. Sci., vol. 2627, Springer, 2002, pp. 15–30.
[25] A.M. Frisch, P.J. Stuckey, The proper treatment of undefinedness in constraint languages, in: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, CP 2009, 2009, pp. 367–382.
[26] T. Frühwirth, Constraint Handling Rules, 1st edn., Cambridge University Press, New York, NY, USA, 2009.
[27] T.W. Frühwirth, Constraint handling rules – what else?, in: Rule Technologies: Foundations, Tools, and Applications, RuleML 2015, 2015, pp. 13–34.
[28] Gecode Team, Gecode: generic constraint development environment, available from http://www.gecode.org, 2016.

[29] I.P. Gent, Arc consistency in SAT, in: Proceedings of the 15th European Conference on Artificial Intelligence, ECAI 2002, 2002, pp. 121–125.
[30] I.P. Gent, C. Jefferson, I. Miguel, Minion: a fast scalable constraint solver, in: Proceedings ECAI 2006, 2006, pp. 98–102.
[31] I.P. Gent, I. Miguel, A. Rendl, Tailoring solver-independent constraint models: a case study with Essence′ and Minion, in: Proceedings of the Seventh Symposium on Abstraction, Reformulation, and Approximation, SARA 2007, 2007, pp. 184–199.
[32] F.S. Hillier, G.J. Lieberman, Introduction to Operations Research, 9 edn., McGraw-Hill, 2010, International edition.
[33] W.J. van Hoeve, G. Pesant, L.M. Rousseau, A. Sabharwal, Revisiting the sequence constraint, in: Proc. 12th Principles and Practice of Constraint Programming, CP 2006, 2006, pp. 620–634.
[34] M. Järvisalo, M.J. Heule, A. Biere, Inprocessing rules, in: Automated Reasoning, 6th International Joint Conference, IJCAR 2012, Springer, 2012, pp. 355–370.
[35] C. Jefferson, N. Moore, P. Nightingale, K.E. Petrie, Implementing logical connectives in constraint programming, Artif. Intell. 174 (2010) 1407–1429.
[36] P. Laborie, P. Refalo, P. Shaw, Model presolve, warmstart and conflict refining in CP Optimizer, http://cp2013.a4cp.org/node/99, 2013, talk given at the CPSOLVERS-2013 workshop at CP2013.
[37] C. Lecoutre, S. Cardon, J. Vion, Second-order consistencies, J. Artif. Intell. Res. 40 (2011) 175–219.
[38] K. Leo, C. Mears, G. Tack, M.G. de la Banda, Globalizing constraint models, in: Proceedings of the 19th International Conference of the Principles and Practice of Constraint Programming, CP 2013, 2013, pp. 432–447.
[39] K. Leo, G. Tack, Multi-pass high-level presolving, in: Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI, 2015, pp. 346–352.
[40] J. Marques-Silva, Practical applications of boolean satisfiability, in: 9th International Workshop on Discrete Event Systems, WODES 2008, 2008, pp. 74–80.
[41] C. Mears, M.G. de la Banda, Towards automatic dominance breaking for constraint optimization problems, in: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, 2015, pp. 360–366.
[42] C. Mears, T. Niven, M. Jackson, M. Wallace, Proving symmetries by model transformation, in: Proceedings of the 17th International Conference on the Principles and Practice of Constraint Programming, CP 2011, 2011, pp. 591–605.
[43] P. Meseguer, C. Torras, Exploiting symmetries within constraint satisfaction search, Artif. Intell. 129 (2001) 133–163.
[44] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings of the 38th annual Design Automation Conference, ACM, 2001, pp. 530–535.
[45] P. Nightingale, CSPLib problem 057: Killer Sudoku, http://www.csplib.org/Problems/prob057.
[46] P. Nightingale, The extended global cardinality constraint: an empirical survey, Artif. Intell. 175 (2) (2011) 586–614.
[47] P. Nightingale, Ö. Akgün, I.P. Gent, C. Jefferson, I. Miguel, Automatically improving constraint models in Savile Row through associative–commutative common subexpression elimination, in: 20th International Conference on Principles and Practice of Constraint Programming, CP 2014, Springer, 2014, pp. 590–605.
[48] P. Nightingale, Ö. Akgün, I.P. Gent, C. Jefferson, I. Miguel, P. Spracklen, Automatically Improving Constraint Models in Savile Row: Supplementary Material, Tech. rep., University of St Andrews, 2017, available from https://pn.host.cs.st-andrews.ac.uk/savilerowjournal-supplement.pdf.
[49] P. Nightingale, A. Rendl, Essence' description, 2016, arXiv:1601.02865 [cs.AI].
[50] P. Nightingale, P. Spracklen, I. Miguel, Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row, in: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming, CP 2015, 2015, pp. 330–340.
[51] B.D. Parrello, W.C. Kabat, L. Wos, Job-shop scheduling using automated reasoning: a case study of the car-sequencing problem, J. Autom. Reason. 2 (1) (1986) 1–42.
[52] S. Prestwich, CSPLib problem 028: balanced incomplete block designs, http://www.csplib.org/Problems/prob028.
[53] S.D. Prestwich, CNF encodings, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, in: Front. Artif. Intell. Appl., vol. 185, IOS Press, 2009, pp. 75–97.
[54] J.F. Puget, Constraint programming next challenge: simplicity of use, in: Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP 2004, 2004, pp. 5–8.
[55] J.F. Puget, Symmetry breaking using stabilizers, in: F. Rossi (Ed.), CP, in: Lect. Notes Comput. Sci., vol. 2833, Springer, 2003, pp. 585–599.
[56] J.C. Régin, Generalized arc consistency for global cardinality constraint, in: Proceedings AAAI 1996, 1996, pp. 209–215.
[57] J.C. Régin, Cost based arc consistency for global cardinality constraints, Constraints 7 (3–4) (2002) 387–405.
[58] J.C. Régin, J.F. Puget, A filtering algorithm for global sequencing constraints, in: Proc. 3rd Constraint Programming, CP 97, 1997, pp. 32–46.
[59] A. Rendl, Effective Compilation of Constraint Models, Ph.D. thesis, University of St Andrews, 2010.
[60] A. Rendl, I. Miguel, I.P. Gent, C. Jefferson, Automatically enhancing constraint model instances during tailoring, in: V. Bulitko, J.C. Beck (Eds.), SARA, AAAI, 2009.
[61] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier, 2006.
[62] Y. Shang, B.W. Wah, A discrete lagrangian-based global-search method for solving satisfiability problems, J. Glob. Optim. 12 (1) (1998) 61–99.
[63] J.P.M. Silva, I. Lynce, Towards robust CNF encodings of cardinality constraints, in: Proc. 13th Principles and Practice of Constraint Programming, CP 2007, 2007, pp. 483–497.
[64] B.M. Smith, Symmetry and search in a network design problem, in: 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2005, 2005, pp. 336–350.
[65] P.J. Stuckey, G. Tack, MiniZinc with functions, in: Proceedings of 10th International Conference on Integration of AI and OR Techniques, 2013, pp. 268–283.
[66] S. Subbarayan, D.K. Pradhan, NiVER: non increasing variable elimination resolution for preprocessing SAT instances, in: Eighth International Conference on Theory and Applications of Satisfiability Testing, SAT 2005, 2005, pp. 276–291.
[67] N. Tamura, A. Taga, S. Kitagawa, M. Banbara, Compiling finite linear CSP into SAT, Constraints 14 (2) (2009) 254–272.
[68] P. Van Hentenryck, The OPL Optimization Programming Language, MIT Press, Cambridge, MA, USA, 1999.
[69] T. Walsh, SAT v CSP, in: Proc. 6th International Conference on the Principles and Practice of Constraint Programming, in: Lect. Notes Comput. Sci., vol. 1894, Springer, 2000, pp. 441–456.
[70] Y. Yan, C. Gutierrez, J.C. Jeriah, F.S. Bao, Y. Zhang, Accelerating SAT solving by common subclause elimination, in: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI 2015, 2015, pp. 4224–4225.