The Cost of Flattening with Common Subexpression Elimination

Andrea Rendl, Ian Miguel and Ian P. Gent

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

Abstract. Compiling a solver-independent constraint model to solver input usually involves *flattening*, the decomposition of complex constraints into simpler expressions to suit the solver, introducing additional variables and constraints. In previous work [8], we have proposed extending flattening with common subexpression elimination (CSE) which can reduce the overhead introduced during flattening. In this work, we formally analyse the *cost* of standard flattening and CSE-based flattening: we compare time and space complexity and investigate the potential variable and constraint reduction from CSE-based flattening, its scope and limitations. Furthermore, we discuss flattening whole problem *classes* and show how to integrate CSE into class-wise flattening. We highlight the differences to *instance*-wise flattening and discuss open questions. Finally, our empirical analysis confirms our theoretical findings and demonstrates the benefits of CSE-based flattening.

1 Introduction

Solver-independent constraint modelling languages have become increasingly popular since they provide high-level constructs that facilitate modelling. However, compiling a high-level constraint model to solver input usually involves *flattening*, the decomposition of complex constraints into simpler expressions that conform to the target solver's propagators, a procedure that introduces additional variables and constraints. In previous work [8], we have proposed extending flattening with common subexpression elimination (CSE) which can reduce the overhead introduced during flattening.

In this work we extend our previous work with two contributions: First, we theoretically analyse the cost of standard flattening and CSE-based flattening, comparing time and space complexity, investigating the potential variable and constraint reduction from CSE-based flattening, its scope and limitations (Section 3).

Second, we extend the investigation of flattening *instances* to flattening *problem classes* (Section 4). We show how to integrate CSE into class-wise flattening and present simple reformulations to increase the number of common subexpressions. We highlight differences to instance-wise flattening, and show that problem-wise flattening can be an interesting alternative to flattening problems instance-wise. Related Work is discussed in Section 5.

Finally, we perform an empirical analysis on a selection of problems that confirm the benefits of common subexpression elimination derived from our theoretical analysis, and illustrate the different benefits of instance-wise and class-wise flattening.

2 Background

A constraint problem *instance* is defined in terms of a constraint satisfaction problem (CSP). Typically, a CSP is an instantiation of a whole problem *class* which is a parameterised constraint problem model (e.g. the 4-queens instance is an instantiation of the n-queens problem class). Solver-independent constraint modelling languages, such as OPL [12], MiniZinc [18] or ESSENCE$'$ [7], provide means to formulate both problem classes and instances. Usually, a problem class formulation is paired with a parameter specification to create an instance.

Throughout this paper, we consider constraints as expression trees, where nodes correspond to n-ary operators (or global constraints), node's children represent arguments, and leaves are variables/constants. In many solvers, constraints are implemented as *propagators* which are 'granular' constraints that take only variables as arguments[1] (i.e. their expression tree depth is 1). Therefore, in many cases, a target solver does not directly support expressions as they are formulated in a rich, solver-independent modelling language. Thus, such expressions need to be decomposed into a conjunction of simpler expressions that conform to the constraints provided by the solver. This decomposition is generally known as *flattening*.

A *flat representation* E'_S of expression tree E wrt target solver S is a conjunction of simple expression trees, defined as $E'_S{:=}\wedge_i E'_i$, where for each conjoint expression E'_i there exists a propagator p in solver S that matches the tree structure of E'_i, and $E'_s \equiv E$. We assume that prior to flattening, every expression tree has been preprocessed such that its tree structure conforms to the propagators provided by solver S. In other words, for every node N in E, there exists a propagator in solver S that corresponds to operation N and has the same arity as node N has children (e.g. if E contains a sum-node with n children, then solver S must have an n-ary sum propagator)[2]. If all constraints in a model M are adapted to solver S in this way, we will refer to it as M_S.

3 Flattening Constraint Instances

In this section we formally investigate how to enhance standard instance flattening with common subexpression elimination (CSE) [8]. Starting from a typical flattening approach, we extend it with CSE and compare the differences in time and space complexity, as well as the resulting instance sizes.Finally, we discuss how to increase the number of common subexpression in an instance by reformulation.

3.1 A Typical Flattening Approach

We summarise a wide-spread flattening approach, that, given a constraint instance M_S, produces a flat instance M'_S. We specifiy the basic algorithm, *flattenInstance*, in Algorithm 1, which employs a helper procedure, *flatten* that is summarised in Algorithm 2.

[1] Exceptions are solvers such as Eclipse Prolog, that flatten their input internally.

[2] Note, that this preprocessing procedure can be embedded into flattening, but has been left out in this discussion to not obscure the task of flattening.

Algorithm 1 .*flattenInstance*(M_S) flattens constraint instance M_S to flat instance M'_S.

```
Require: $M_S$: problem instance
 1: global flatConstraints, constraintBuffer, auxVars ← empty lists
 2: for all $E \in M_S$.constraints do
 3:     constraintBuffer ← empty
 4:     $E'_0$ ←flatten($E$ false)
 5:     $E'_S \leftarrow E'_0 \wedge (\bigwedge_i E'_i \in$ constraintBuffer)
 6:     flatConstraints.add($E'_S$)
 7: $M'_S$.constraints ← flatConstraints
 8: $M'_S$.vars ← $\{M_S$.vars $\cup$ auxVars$\}$
 9: return $M'_S$
```

flattenInstance applies *flatten* on each expression in instance M_S (line 4 in Alg. 1). *flatten* is a recursive procedure that iterates over the expression tree in a bottom up fashion. It replaces all nodes N_i in E (except the root node) with an auxiliary variable *Aux*$_i$ and generates the constraint '*Aux*$_i$ = N_i' that connects every auxiliary variable with its corresponding subtree (line 7-9 in Alg. 2). After flattening each subnode of an expression E, *flatten* returns expression E'_0 and the global list *constraintBuffer* contains the k constraints, E'_1,..,E'_k, of the form '*Aux*$_i$=E_i'. The flat representation E'_S of expression E is constructed by combining E'_0 with these constraints: $E'_S = E'_0 \wedge E'_1 \wedge \ldots \wedge E'_k$ (line 5 in Alg. 1). As an example, consider Figure 1 that illustrates how *flatten* decomposes the expression $a \Rightarrow ((x<3) \wedge (y>5) \wedge (z=0))$ into the flat representation

$$\begin{aligned} & a \Rightarrow aux4 \\ \wedge\ & aux4 \leftrightarrow (aux1 \wedge aux2 \wedge aux3) \\ \wedge\ & aux1 \leftrightarrow (x<3) \wedge aux2 \leftrightarrow (y>5) \wedge aux3 \leftrightarrow (z=0) \end{aligned}$$

After generating the flat representation E'_S for each constraint E in instance M_S, *flattenInstance* constructs the flat instance M'_S, whose variables consist of M_s's decision variables combined with the auxiliary variables, and whose constraints are the flattened expressions (line 7-8 in Alg. 1).

Algorithm 2 . ***flatten***(E,*flatten2Aux*) recursive procedure that flattens expression tree E, where *flatten2Aux* denotes if E will be flattened to an auxiliary variable.

```
Require: $E$ : expression tree, flatten2Aux : Boolean
 1: if ¬(all of $E$'s children are leaves) then
 2:     for all $e_i \in$ children($E$) do
 3:         if ¬($e_i$.isLeaf) then
 4:             $e'_i$ ← flatten($e_i$, true)
 5:             $E$.replaceChildWith($e_i$,$e'_i$)
 6: if flatten2Aux then
 7:     Aux← createNewVariable($E$.lb, $E$.ub); auxVars.add(Aux)
 8:     constraintBuffer.add('Aux = $E$')
 9:     return Aux
10: else
11:     return $E$
```

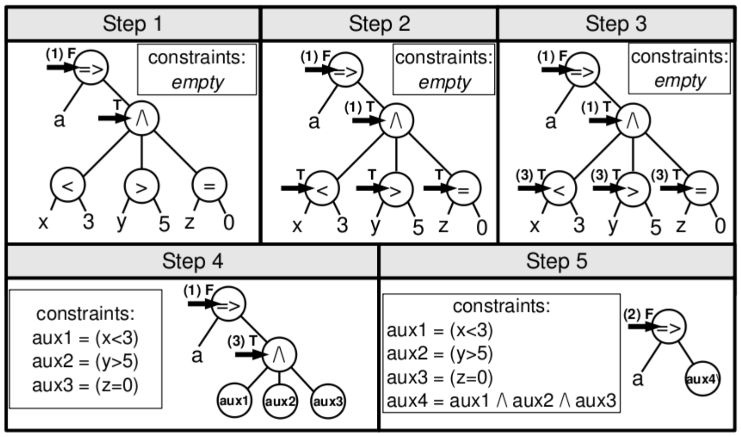

Fig. 1. **Example: Flattening** $a \rightarrow ((x{<}3) \wedge (y{>}5) \wedge (z{=}0))$ using *flatten* (Alg. 2). Arrows pointing at nodes denote that *flatten* has been invoked on the subtree. **T** and **F** represent the Boolean value for *flatten2Aux*. The numbers represent parts of Algorithm 2: **(1)** invoke *flatten* on non-leaf children (line 4-5), **(2)** return expression (line 10-11), **(3)** flatten to auxiliary variable (line 7-9). **Step 1**: the expression tree has a child that is not a leaf (conjunction), hence *flatten* is invoked on it. **Step 2**: the conjunction, has non-leaf children, so *flatten* is invoked on them. **Step 3**: the conjunction's children have leaves as children that will be flattened to a variable (*flatten2Aux=true*). **Step 4**: auxiliary variables *aux1*,*aux2*, *aux3* are created and replaced for the conjunction's children; now the conjunction will be flattened to a variable. **Step 5**: *aux4* represents the conjunction and *flatten* returns the flattened expression $a \Rightarrow$ *aux4*.

We can easily see, that *flattenInstance* will generate a valid flat instance M'_S. *flatten* is applied to every constraint E of M_S, which is recursively applied to all subnodes of E, creating an auxiliary variable for each. Since we assume that every node in E corresponds to a propagator in target solver S, it is sufficient to replace each subnode that is not a leaf with an auxiliary variable to conform to the target solver.

Lemma 1. *If constraint instance M_S contains n constraints that contain m nodes in their expression trees (with $m \geq n$), then* flattenInstance *will generate flat instance M'_S with m constraints and $m - n$ auxiliary variables.*

Proof. *flattenInstance* applies *flatten* to every constraint E in M_S (line 4 in Alg. 1). *flatten* is recursively invoked on every subnode that is not a leaf (line 4 in Alg. 2), and creates an auxiliary variable and constraint for each node for which *flatten2Aux=true* (line 4 in Alg. 2). This holds for every node in E, except the root node (*flatten2Aux=false* only for the root node, see line 4 in Alg. 1). Therefore, *flatten* creates one auxiliary variable and one '*Aux*=E'-constraint for every node, except the root nodes. For every root node, it returns one constraint (line 11 in Alg. 2). In summary, *flatten* generates $m - n$ auxiliary variables and m constraints ($m - n$ '*Aux*=E'-constraints and n root node constraints), where n is the number of constraints (i.e. root nodes) and m the number of nodes in M_S. □

As an example, consider the constraint instance consisting of the constraint from Fig. 1: $a \Rightarrow ((x{<}3) \wedge (y{>}5) \wedge (z{=}0))$. M_S contains 1 constraint (n=1) and 5 nodes (m=5), therefore *flatten* generates M'_S containing 5 constraints using 5-1 auxiliary variables.

Tight Bounds for Auxiliary Variables An important issue in flattening is deriving tight bounds for auxiliary variables. In *flatten*, the procedure *createNewVariable* creates an auxiliary variable with lower bound lb and upper bound ub (line 7 in Alg. 2). lb and ub are obtained from node E: we assume that every node has the constant attributes lb and ub which are computed in a bottom-up fashion on the expression tree before flattening. Since leaves are either constants or variables that range over a finite domain, we can compute lb and ub for each node in the expression tree[3].

3.2 Extending Flattening with Common Subexpression Elimination(CSE)

In the previous section we have seen a typical flattening procedure, *flattenInstance*, that invokes *flatten* on every constraint expression of the instance. Note, that if two (or more) constraints have identical subtrees (*common subexpressions*) then *flatten* will *not* exploit this equivalence, which results in three redundancies: First, *flatten* creates a different auxiliary variable for each subtree, while each identical subtree could/should be represented by the same auxiliary variable. Second, Creating redundant auxiliary variables also creates redundant constraints to initialise these variables. Third, *flatten* is *repeating* work that it has already performed. Consequently, extending *flatten* to detect and exploit common subexpressions so as to eliminate these redundancies is desirable.

We extend the general flattening procedure *flattenInstance* with common subexpression elimination (CSE) in the new procedure *flattenInstance*$_{CSE}$, summarised in Algorithm 3. *flattenInstance*$_{CSE}$ employs a *hashMap* that maps each flattened node (i.e. subexpression) to its corresponding auxiliary variable. The hashmap is used by the new *flatten*-procedure, *flatten*$_{CSE}$, to detect identical subexpressions that have been previously flattened. *flatten*$_{CSE}$ is an extension of *flatten*, and is summarised in Algorithm 4 where extensions are highlighted in in red font. The extensions are limited to line 1-9 in Alg. 4, and can be summarised as follows: whenever we need to flatten a non-leaf child e_i of current node E, we look for an entry of e_i in *hashmap* (line 4). If there is an entry, we re-use the auxiliary variable to which e_i is mapped (line 5), instead

[3] lb and ub can be computed as a synthesised attribute on the syntax tree

Algorithm 3 ***flattenInstance***$_{CSE}(M_S)$ flattens constraint instance M_S to M'_S with CSE. Differences/extensions to Algorithm 1 are given in *red font*.

```
Require: $M_S$: problem instance
 1: global flatConstraints, constraintBuffer, auxVars ← empty lists
 2: global hashMap ← empty hashmap
 3: for all $E \in M_S$.constraints do
 4:     constraintBuffer ← empty
 5:     $E'_0 \leftarrow$ flatten$_{CSE}(E, false)$
 6:     $E'_S \leftarrow E'_0 \wedge (\bigwedge_i E'_i \in$constraintBuffer$)$
 7:     flatConstraints.add($E'_S$)
 8: $M'_S$.constraints ← flatConstraints
 9: $M'_S$.vars ← $\{M_S$.vars $\cup$ auxVars$\}$
10: return $M'_S$
```

Algorithm 4 *flatten*$_{CSE}$(E,*flatten2Aux*) recursive procedure based on Algorithm 2. Extensions are given in red font.

```
Require: $E$ : expression tree, *flatten2Aux* : Boolean flattened to aux var
 1: if ¬ (all of $E$'s children are leaves) then
 2:     for all $e_i \in children(E)$ do
 3:         if ¬($e_i$.isLeaf) then
 4:             if $hashMap.contains(e_i)$ then
 5:                 $e_i' \leftarrow hashMap.get(e_i)$
 6:             else
 7:                 $e_i' \leftarrow flatten_{CSE}(e_i, S, true)$
 8:                 $hashMap.add(e_i, e_i')$
 9:             $E.replaceChildWith(e_i, e_i')$
10: if flatten2Aux then
11:     $Aux \leftarrow createNewVariable(E.\mathrm{lb}, E.\mathrm{ub})$; auxVars.add(Aux)
12:     constraintBuffer.add('$Aux = E$')
13:     return $Aux$
14: else
15:     return $E$
```

of flattening e_i again. Otherwise (i.e. if there is no match in *hashmap*), we flatten e_i to auxiliary variable e_i' (line 7) and add $e_i \rightarrow e_i'$ to *hashmap* (line 8).

We can easily see, that *flattenInstance*$_{CSE}$ will flatten each *unique* subnode N exactly once. If the number of unique nodes m_u is less than the number of nodes m in instance M_S, then M_S contains common subexpressions.

Theorem 1. *If constraint instance M_S contains n constraints that contain m subexpressions of which m_u are unique (with $m \geq m_u \geq n$), then* flattenInstance$_{CSE}$ *will generate a flat instance M_S' with $m_u - n$ auxiliary variables and m_u constraints.*

Proof. *flattenInstance*$_{CSE}$ applies *flatten*$_{CSE}$ to every constraint E in M_S (line 5 in Alg. 3). If E has a non-leaf child e_i (a subnode), then there are two different cases (line 4 in Alg. 4): First, if there is no entry of e_i in *hashmap*, e_i is flattened to auxiliary variable e_i' and e_i and e_i' are added to *hashmap* (line 7-8) Hence, if e_i has to be flattened again in M_S, there will be an entry in *hashmap*. Second, if there is an entry of e_i in *hashmap*, (e_i must have been flattened before), the corresponding auxiliary variable e_i' is retrieved from *hashmap* (line 5). Therefore, *flatten*$_{CSE}$ is only invoked on those children that have not been previously flattened, i.e. every unique node is flattened exactly once.

This results in one auxiliary variable and one '*Aux*=E'-constraint for every unique node that is not a root node, and one constraint for every unique root node (see Lemma 1). In summary, *flattenInstance*$_{CSE}$ will generate a flat instance M_S' that contains $m_u - n$ auxiliary variables (one for each unique node, minus the root nodes) and m_u constraints ($m_u - n$ '*Aux*=E'-constraints and n root-node-constraints), where m_u is the number of unique nodes, and n the number of constraints in M_S. □

3.3 Comparing Standard Flattening with CSE-based Flattening

We analyse the differences of applying *flattenInstance* and *flattenInstance*$_{CSE}$ on problem instance M_S with n constraints and m subexpressions of which m_u are unique. See our Experimental Section (Section 6) for an empirical analysis.

Flat Instance Size

Applying *flattenInstance* and *flattenInstance*$_{CSE}$ on M_S will yield the flat instances $M'_{S,1}$ and $M'_{S,2}$ respectively. $M'_{S,1}$ will contain m constraints and $m - n$ auxiliary variables (Lemma 1), and $M'_{S,2}$ will contain m_u constraints and $m_u - n$ auxiliary variables (Theorem 1). If $m_u < m$, i.e. M_S contains common subexpressions, then $M'_{S,2}$ will contain $m - m_u$ less constraints and auxiliary variables than $M'_{S,1}$. Otherwise, if $m = m_u$, both flat instances are identical.

Space Complexity

flattenInstance employs the lists *flatConstraints*, *auxVars* and *constraintBuffer*, as well as the representation of constraint model M_S and M'_S. All these datastructures require a maximal capacity of m, where m is the number of subexpressions in M_S (Lemma 1). Therefore the space complexity of *flattenInstance* lies in $O(m)$.

flattenInstance$_{CSE}$ uses the same data structures as *flattenInstance*, with the addition of the hashmap to store flattened subexpressions. The hashmap is String-based, i.e. each expression tree and auxiliary variable is stored as a String (instead of as a tree), which facilitates matching. It stores the m_u unique nodes and the corresponding auxiliary variables, thus the hashmap uses $2 * m_u *$*sizeof*(String) units of memory. Therefore, *flattenInstance*$_{CSE}$ uses $2 * m_u *$*sizeof*(String) more units of memory than *flattenInstance*. Note, that this is a constant increase, since $m_u \leq m$, and therefore the space complexity still lies in $O(m)$.

Time Complexity

Let f be the number of operations that are required to flatten a node of an expression. From Lemma 1 we know that *flattenInstance* performs these operations m times, since it is applied to every node in M_S. Therefore, *flattenInstance* has a runtime of $f * m$ which lies in $O(m)$.

flattenInstance$_{CSE}$ adds two instructions to the flattening process: the hashmap check followed by either retrieving an object from the hashmap or creating an entry to the hashmap. Since operations on hashmaps are in $O(1)$ on average, the number of operations for flattening in *flatten*$_{CSE}$ is f+2 operations. From Theorem 1 we know that *flattenInstance*$_{CSE}$ flattens all m_u unique nodes/subexpression exactly once, hence the overall sum of operations in *flattenInstance*$_{CSE}$ is m_u*(f+2).

Therefore, if m_u=m, i.e. M_S has no common subexpressions, *flattenInstance*$_{CSE}$ has a runtime of m*(f+2), which, compared to m*(f) of *flattenInstance* is a constant increase in runtime that lies in $O(m)$. However, if m_u<m, i.e. there exist common subexpressions in M_S, the runtime is (m−(m-m_u))*(f+2) where m-m_u>0, hence the runtime is less than linear in m, since it is in O($m - c(m)$) where $c(m) = m - m_u$.

3.4 Reducing the Number of Unique Nodes m_u

In the previous section we have seen that flattening an instance M_S with CSE yields a flat constraint instance M'_S with $m - m_u$ less constraints and auxiliary variables than applying flattening without CSE, while also reducing the flattening runtime by m-m_u. Naturally, we are interested in maximising our benefits: if we reduce m_u, i.e. we increase the number of common subexpressions, the reduction in instance size and runtime, $m - m_u$, increases.

The number of unique nodes m_u in an instance M_S is can be decreased by reformulating equivalent, but not identical subtrees into a common tree structure. For instance, if we have two equivalent (but not identical) subtrees $a*(b+c)$ and $a*b+a*c$, then we can reformulate the latter to the representation of the first, yielding two identical trees $a * (b + c)$, which decreases m_u if performed for every $a * b + a * c$ in the instance.

To apply reformulations in a structured, efficient way, a *detection* step is required, where two (or more) equivalent but not identical subtrees are spotted in an instance. Naturally, the effort invested to detect these equivalences can be arbitrarily large, especially for very powerful reformulations. As an example, consider detecting a clique of disequalities (which is NP-comeplete) to match global constraint *alldifferent*. Therefore, we restrict ourselves to two basic measures that involve little detection and little reformulation, but have strong potential for reducing m_u.

Normalisation An obvious measure to reduce m_u is to *normalise* every expression tree: many operators, such as conjunction or addition, are commutative, hence there are ambigious representations of such nodes. As an example, $a \wedge b$ and $b \wedge a$ are an equivalent expression, but their expression trees are not identical. By imposing an *order* on the arguments of all commutative operators, such ambigiouties are eliminated, without requiring a detection step. Furthermore, constant *evaluation* can reduce the number of unique nodes, by reducing subtrees that consist entirely of constants. As an example, $2 * 6$ and $3 * 4$ are both evaluated to the same leaf 12. Constant evaluation is an inexpensive procedure that again requires no detection step. Note that normalisation is a common procedure during preprocessing before flattening in many systems [8, 18].

Negation Reformulation The negation reformulation is concerned with reformulating a subtree to its negated form in order to detect identical subtrees, and has first been proposed in [8]. The idea is to include the detection step into flattening of expression tree E: whenever a non-leaf child e_i of E has no entry in the hashmap, the hashmap is consulted with $\neg e_i$. If there is a match, that returns auxiliary variable *Aux*, then e_i is replaced by the negated auxiliary variable $\neg$*Aux*. Note that this reformulation is only valid, if the target solver S allows negated leaves as arguments in its constraints. Fig. 2 illustrates the idea on an example.

The negation reformulation is directly embedded into the recursive flattening procedure, adding two hashmap operations and the reformulation of the corresponding node[4]. Therefore we restrict the detection step to be performed only on relational operators (i.e. $=$, $<$, etc), since they are particularily promising candidates for this reformulation.

[4] Note that for reasons of space we cannot include an algorithm for the negation reformulation.

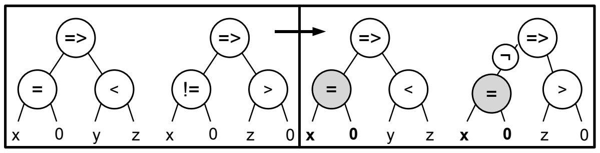

Fig. 2. Negation Reformulation Example: Flattening $(x{=}0) \Rightarrow (y{=}z)$ and $(x! = 0) \Rightarrow (y{>}0)$ with negation reformulation, creating identical subtrees (x=0)

4 Flattening Problem Classes

In this section, we discuss how to extend our approach of flattening *instances* to flattening problem *classes*. Most flattening tools are limited to flattening instances, and flattening of constraint problem classes has not been thoroughly investigated. There are two main reasons for flattening problem classes: first, to support solvers that take problem classes as input: solvers such as Gecode [5], Eclipse [4] or Choco [2] are libraries of programming languages, where problems are formulated as programs and parameters can be specified at runtime. Furthermore, for many library-based solvers the compilation of large problem models is infeasible - compact parameterised problems are much quicker to compile. Second, it can be more time-efficient to perform flattening and enhancement techniques *once* on problem class level instead of *several times*, for every problem instance.

In the following, we show how to extend instance-wise flattening to class-wise flattening: first we extend the instance-based expression tree structure so as to support parameterised expressions. Then, we extend the flattening procedure to deal with parameterised expressions. Finally, we discuss common subexpression elimination (CSE) on problem class level.

4.1 Representing Parameterised Expressions

Parameters can occur in different parts of the model which requires extensions to the expression tree representation from the instance-level.

Parameters in Bounds of Expression Tree Nodes. Parameters can appear as leaves in expressions trees, which then have no specified, constant domain. Therefore, we extend the attributes lb and ub (that represent lower and upper bound of each node) to contain parameters, and state that if a parameter k appears as leaf in an expression tree, then its lower and upper bound is defined as $k..k$. Similarily, for parameters used as lower and/or upper bounds of a decision variable's domain, we state that if parameters k_1, k_2 scale the domain of a decision variable x and x appears as leaf in an expression tree E, then x's lower and upper bound is defined as $k_1..k_2$ under the assumption that $k_1 \leq k_2$.

Parameters In Quantifications. Parameters are often used to scale quantifications, such as parameter n in the example **forall** i : int(1..n) . x[i] $\neq$ y[i]. In a problem instance, n would be specified and we would unroll the quantification, generating a set of disequality constraints. However, n is not specified, the quantification cannot be unrolled, thus we need to define the notion of a quantified expression as part of our expression tree representation.

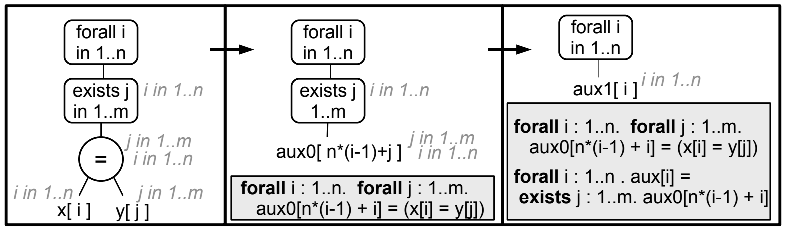

Fig. 3. Class Flattening Example: Flattening $\forall i{\in}1..n.\ \exists j{\in}1..m.\ x[i] = y[j]$, grey labels represent the quantifier attributes; constraints in the grey box show the flattened expressions.

We extend the expression trees from instance-level to support quantifiers, particularly $\forall$, $\exists$ and $\sum$. Note that these quantifiers are only allowed in the expression tree if the target solver supports n-ary conjunction ($\forall$), n-ary disjunction ($\exists$) and n-ary addition ($\sum$). Each quantification node N has one argument: the quantified expression. The quantifying variable(s) and the corresponding quantified domain(s) is(are) stored as attributes in N. These attributes are also included into all subnodes and leaves that are quantified. Thus each node/leaf in an expression tree 'knows' if (and how) it is quantified (as an example, see Fig. 3 where the attributes are illustrated as grey labels).

4.2 Flattening Quantified Subexpressions

The challenge in flattening problem classes is flattening quantified subexpressions that cannot be unrolled because there domain is parameterised. In our approach, a quantified expressions is flattened to an array of auxiliary variables whose size is derived by the domain of the corresponding quantifiers, which is retrieved from the quantification attributes of the quantified subexpression. For illustration, consider '**exists** i : int($1..n$) . x[i]=i'. The quantified subexpression 'x[i]=i' is flattened to a Boolean auxiliary array *auxArray* of length n (since i ranges from $1..n$)[5], resulting in the flat expressions:

forall i : int($1..n$) . *auxArray*[i] $\leftrightarrow$ x[i] = i
exists i : int($1..n$) . *auxArray*[i]

We summarise flattening quantified expressions in Algorithm 5 that shows an excerpt of the recursive flattening procedure where auxiliary variables and constraints are added to the flat instance. Extensions to *flatten*/*flatten*$_{CSE}$ are given in red font: if the flattened expression E is quantified, then it is flattened to an auxiliary variable array (line 4). Note that for convenience, we represent auxiliary variable arrays as 1-dimensional arrays. The length of the array is determined from the lengths of the quantifying variables' domains: if quantified expression E is quantified by k quantifying variables v_i that each range over the domain $lb_i..ub_i$, then *length* of the array is $\prod_{1..k}^{i} ub_i - lb_i + 1$. As an example, in Fig. 3, aux0 represents an expression that is quantified by $i \in 1..n$ and $j \in 1..m$ and hence has length $(m - 1 + 1) * (n - 1 + 1)$, i.e. $n * m$.

After creating the auxiliary array, a constraint is generated to link the array with the subtree E (line 5). This constraint is composed of one universal quantification for each quantifying variable. The *index* of the auxiliary array is computed from the quantifier attributes and are adapted for 1-dimensional arrays (for an example, see Fig. 3).

[5] We derive the 'length' of each quantifier domain ($lb..ub$) from ub-lb+1

Algorithm 5 Excerpt of ***flattenClass***$_{CSE}$(E,*flatten2Aux*) for flattening problem classes, based on Alg. 4. Extensions are given in red font.

```
Require: $E$ : expression tree, flatten2Aux : Boolean flattened to aux var
 1: ....
 2: if flatten2Aux then
 3:   if $E$ is quantified then
 4:     AuxArray = createNewVarArray($E$.lb,$E$.ub, $E$.qt.length); vars.add('AuxArray')
 5:     constraints.add('{'$\forall x \in x$.dom' | $x \in E$.qt.vars } . AuxArray[$E$.qt.index] = $E$')
 6:     return AuxArray[$E$.qt.index]
 7:   else
 8:     Aux ← createNewVariable($E$.lb, $E$.ub); auxVars.add(Aux)
 9:     constraintBuffer.add('Aux = $E$')
10:     return Aux
11: else
12:   return $E$
```

Redundancies from Flattening Quantified Subexpressions In some cases, constraints in quantifications are guarded by a boolean expression, such as $(i\text{!=}j)$ in

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

Flattening will introduce $2{*}n^2$ auxiliary variables (one array of length n^2 for each multiplication), however, since $(i\text{!=}j)$ will evaluate to *false* in n cases, only $2{*}(n^2 - n)$ auxiliary variables are actually used, the rest is unconstrained. Note that this does not occur in instance-wise flattening, since constant evaluation reduces expressions of the form '*false*$\Rightarrow E$' to *true* before flattening, so E is never flattened.

Eliminating this redundancy raises two difficult questions: first, how to generally determine the number of unconstrained auxiliary variables where guard and quantification can be arbitrarily complex? Second, how to best represent an array of auxiliary variables, of which some elements are not used: do we need new data structures or does there exist a general mapping to a simpler data structure? Addressing these questions is an important part of our future work. For now, our investigations have shown that the introduced redundancy only matters if the auxiliary variables are included into search, otherwise the impact is marginal (for the examples we have considered).

4.3 Common Subexpression Elimination on Problem Class Level

Common subexpression elimination (CSE) as described for problem instances is directly applicable to problem classes. However, it has to be done with care: consider, for example, two nodes 'x[i]*y[i]' and 'x[i]*y[i]' that occur in different expressions trees. They are identical, but since i might range over different domains in the trees, they are not necessarily equivalent. Therefore, we need to *normalise* quantifying variables before applying CSE-based flattening.

Normalising Quantifying Variables has two aims: first, to rename quantifying variables if they have been defined over a different domain somewhere else in the model (eliminating identical subtrees that are not equivalent) and second, to unify quantifying variables that range over the same domain (creating identical subtrees that are equivalent). As an example for the latter, consider the two quantifications below, where both i and j range over $(1..n)$, and re-writing j into i reveals the common leaf x[i].

$$\textbf{forall}\ i : \text{int}(1..n)\ .\ \text{x}[i] \neq i \quad \Big| \quad \textbf{forall}\ j : \text{int}(1..n)\ .\ \text{x}[j] \leq \text{y}[j]$$

Quantifier Normalisation can be achieved by once iterating over all constraints in the constraint model and collecting/matching quantifiers and their domains, which lies in $O(m)$. This can be easily integrated into general normalisation/evaluation.

Reformulating 'Common' Quantifiers In the case when two or more quantifiers are used in the same quantification over the same domain, renaming as described above cannot be performed. Therefore we construct a hashmap during preprocessing, that collects all quantifiers that range over the same domain. This hashmap is used during flattening, when matching for syntactically common subexpressions: whenever a quantified subexpression has no common subexpression, the hashmap is consulted for 'equivalent' quantifiers, and the quantified subexpression is re-written using such a quantifier. As an example, consider the following two quantifications from the basic n-queens model, stating that no two queens may be in the same NW-SE diagonal:

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

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

$$\begin{array}{l}\textbf{forall}\ \ i : \text{int}(1..n).\ \ \text{aux0}[i] = \text{queens}[i] + i \\ \textbf{forall}\ \ i, j : \text{int}(1..n).\ \ (i < j) \Rightarrow (\text{aux}[i] \neq \text{aux}[j])\end{array}$$

The impact of eliminating a quantified subexpression depends on the size of the corresponding auxiliary array, which is obviously greater than the impact from eliminating a subexpression on instance level. In the example above, matching 'queens[j]+ j' saved us n variables, since $j \in (1..n)$.

5 Related Work

Common subexpression elimination is an optimisation technique originating from compiler optimisation [3], that has proven to be powerful in several disciplines, such as Satisfiability [16], Model Checking [14], Proof Theory [19] and Numerical CSPs [1].

Exploiting explicit linear equalities in CP has been well studied [11, 15, 17]. As an example, consider the explicit linear equality $x = y$, where x and y are decision variables. If x and y have the same domain, every occurrence of y can be replaced with x (or vice-versa) and y removed from the set of variables. Otherwise, a new variable, ranging over the intersection of x and y's domain, can replace both throughout. Note however, that our work is concerned with the advanced case, where equivalence is not explicitly stated in the expression, but has to be detected.

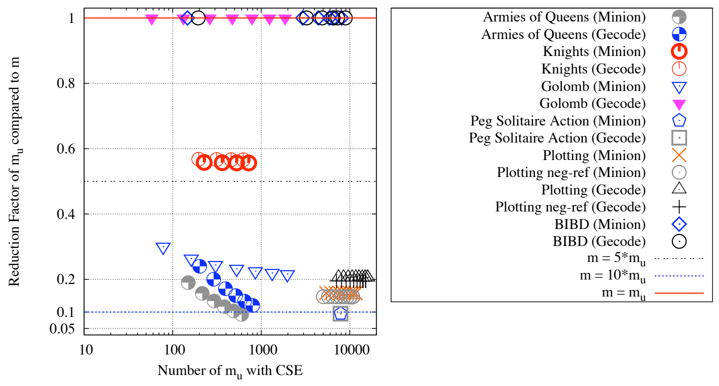

Fig. 4. Instance Reduction. The x-axis represents the number of unique nodes m_u during CSE-based flattening, the y-axis shows the reduction factor compared to m, hence points below y=1 depict the cases when $m_u < m$.

In their work on interval analysis, Schichl *et al* [20, 22] discuss common subexpression elimination in models of mathematical problems, where expressions are represented as directed acyclic graphs (DAG). These studies have much in common with our work, as they examine an alternative approach on eliminating common subexpressions by merging tree nodes in the DAG-representation. Araya *et al* apply Schichl *et al*'s approach to numerical constraint satisfaction problems [1] by transforming the conjunction of all numerical constraints into one DAG that is manipulated so that it contains no more common subexpressions. However, it is not clear if the approach is scalable for large discrete CSPs, which can contain 10,000s of constraints, resulting in an extremely large DAG. Our approach on the other hand, is embedded into the necessary task of flattening, adding constant overhead.

6 Experimental Results

In this section we summarise our empirical evaluation where we investigate standard flattening, CSE-based flattening, and CSE-based flattening extended with the negation reformulation. Note that prior to flattening, expressions are ordered and evaluated. We apply flattening on a selection of problems that have different numbers of common subexpressions. We formulate each problem class in ESSENCE′ [7] and flatten each model to two different target solvers, Minion [6] and Gecode [5], using TAILOR [7], that performs all three different flattening approaches on instance and problem class level. We use TAILORv0.3.2 on Java REv1.6.0 and flatten all instances/classes on the same machine, an MacBook Pro 1,1 with Intel Dual Core (1.83 GHz) and 512MB RAM. We are interested in flattening time, flattening memory usage as well as the number of unique nodes m_u compared to the number of all nodes m.

Fig. 4 depicts the instance reduction with CSE-based flattening and the negation reformulation. We can see that in some cases the number of unique nodes m_u is only

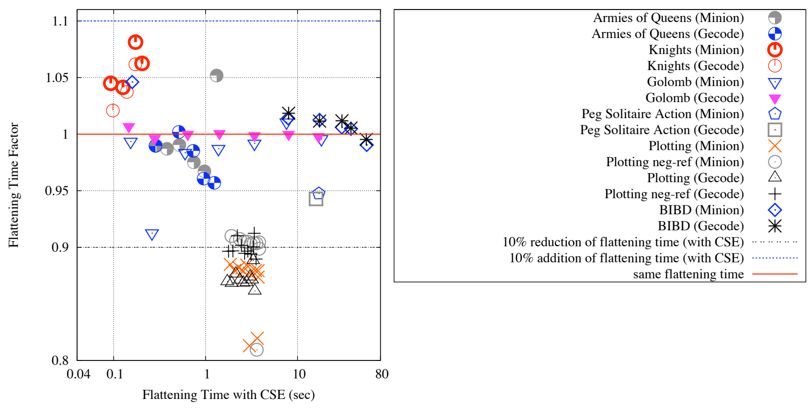

Fig. 5. Flattening Time. The x-axis represents the time for CSE-based flattening, the y-axis shows the factor for comparision with standard flattening: points below y=1 depict cases where flattening time was reduced with CSE, points above when increased.

$\frac{1}{10}$ of the number of all nodes m, hence CSE-based flattening reduces the number of constraints to $\frac{1}{10}$ compared to standard flattening. Furthermore, in Fig. 5 we can see that the flattening time for most instances is *reduced* during CSE-based flattening, hence in many cases, we get smaller instances for less effort in time. In Fig. 6 we compare the memory usage of standard and CSE-based flattening and observe that in many cases the memory usage is reduced to less than 40 % with CSE-based flattening. Note also, that for problems that have no common subexpressions, such as BIBD or the Golomb Ruler (Gecode), we pay no significant overhead for attempting CSE. Empirical results on solving time speed-ups due to CSE can be found in our previous work [8].

7 Conclusion & Discussion

In this paper, we have given a substantial theoretical analysis of the benefits of integrating common subexpression elimination (CSE) into the necessary process of flattening. Our empirical analysis confirms our theoretical findings. Furthermore, we investigate an alternative flattening approach, flattening of whole *problem classes*, and propose an approach of integrating CSE into class-wise flattening.

In our discussion about integrating CSE into flattening, we investigate two simple reformulation and detection techniques to further increase the number of common subexpression in a problem model, which also shows to be successful in our experiments. However, the issue of detection and reformulation raises two difficult questions: First, when does the reduction achieved by reformulation pay off for the efforts invested into detection? Can we predict when an instance is likely to have scope for reducing m_u? Second, is it always obvious which one of two (or more) equivalent representations is more efficient, i.e. provides the strongest propagation in the target solver? For instance, consider again the example of $a * (b + c)$ and $a * b + a * c$: generally, the

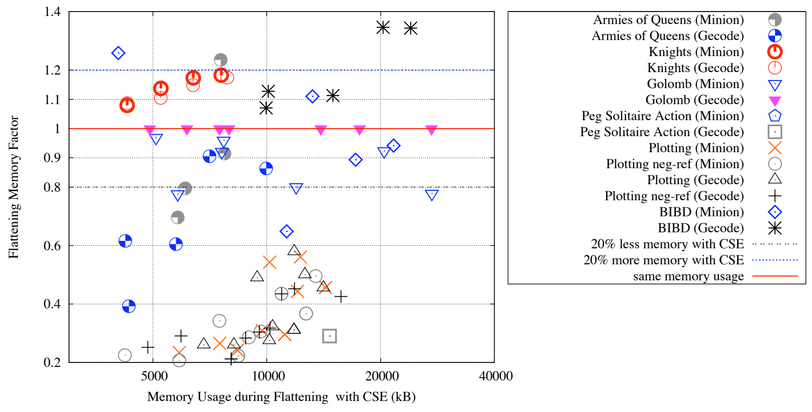

Fig. 6. Memory During Flattening. The x-axis represents the memory for CSE-based flattening, the y-axis shows the factor for comparision with standard flattening: points below y=1 depict cases where memory usage was reduced with CSE, points above when increased.

first representation will provide better propagation, however, what if the instance contains multiple occurrences of the subtrees $a * b$ and $a * c$? We plan to investigate these questions in our future work.

References

1. I. Araya, B. Neveau, G. Trombettoni. Exploiting Common Subexpressions in Numerical CSPS. *in CP*,342-357, 2008.
2. Choco: a java library for constraint programming and explanation-based constraint solving http://choco.emn.fr/
3. J. Cocke, Global common subexpression elimination, *SIGPLAN*,5:20–24,1970.
4. The ECLiPSe Constraint Programming System http://eclipse.crosscoreop.com/
5. Gecode: a Generic Constraint Development Environment http://www.gecode.org
6. I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. *ECAI*, pp 98–102, 2006.
7. I.P. Gent, I. Miguel and A. Rendl. Tailoring Solver-independent Constraint Models: A Case Study with Essence′ and Minion In *SARA*, pp 184-199, 2007.
8. A. Rendl., I. Miguel, I.P. Gent and C. Jefferson Enhancing Constraint Model Instances during Tailoring In *SARA*, 2009, *to appear*
9. A. Rendl., I. Miguel, I.P. Gent and P. Gregory Common Subexpressions in Constraint Models of Planning Problems In *SARA*, 2009, *to appear*
10. I. P. Gent, T. Walsh. CSPLib: A benchmark library for constraints. Technical Report APES-09-1999, 1999.
11. W. Harvey, P. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints*, 7, pp172–203, 2003.
12. P. Van Hentenryck, L. Michel, L. Perron. Constraint programming in OPL *in PPDP*, 1999
13. C. Jefferson, A. Miguel, I. Miguel, A. Tarim. Modelling and Solving English Peg Solitaire. In *Computers and Operations Research* 33(10), pages 2935-2959, 2006.
14. T. Latvala, A. Biere, K. Heljanko and T. A. Junttila Simple Bounded LTL Model Checking. FMCAD,pp 186-200,2004.
15. T. Le Provost, M. Wallace Generalized Constraint Propagation over the CLP Scheme. J. Logic Progr., 16(3), 1993.
16. D. Marinov, S. Khurshid, S. Bugrara, L. Zhang and M. C. Rinard Optimizations for Compiling Declarative Models into Boolean Formulas in *SAT*, pp 187-202, 2005.
17. B.A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n Queens. *IEEE Expert* 5:16-23, 1990
18. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *CP*, LNCS 4741, 529-543, 2007.
19. D.A. Plaisted, and S. Greenbaum A structure-preserving clause form translation, . Jnl of Computation 2,293-304
20. H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization Journal of Global Optimization 33/4 (2005), 541-562
21. B. M Smith, K. Stergiou and T. Walsh. Modelling the Golomb Ruler Problem School of Computing Research Report 1999.12, University of Leeds, June 1999.
22. X.-H. Vu, H. Schichl and D. Sam-Haroud Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems In *ICTAI 2004*, 72-81