

# Effective Compilation of Constraint Models

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

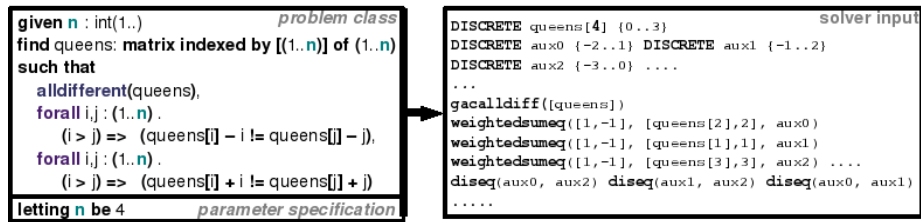
**keywords:** Code Optimisation, Constraint Programming, Automated Modelling

**Abstract.** Compiling solver-independent constraint models to solver input typically involves *flattening*, the decomposition of complex expressions into simpler expressions, introducing additional variables and constraints. In previous work [8], we have informally proposed extending flattening problem instances with common subexpression elimination (CSE), a widespread optimisation technique that has not yet been established in Constraint Programming (CP). This paper extends our previous work with three main contributions. First, we formally analyse the *cost* of flattening instances with CSE, comparing instance reduction and time/space complexity with standard flattening, which outlines its scope. Second, we present how to *increase* the number of common subexpressions in a constraint model by reformulation and include a formal cost analysis. Third, we show how to lift the approach of flattening *instances* to whole problem *classes*, an alternative, novel approach to instance-wise compilation. We formally integrate CSE into class-wise flattening, and show when class-wise compilation is preferable to instance-wise compilation. Finally, experiments confirm our theoretical findings and demonstrate 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) [3] which can reduce the overhead introduced during flattening.

This work extends our previous work with three contributions: First, we analyse the cost of standard flattening and CSE-based flattening, comparing time and space complexity and potential model reductions from flattening with CSE (Section 3). Second, we investigate reformulations to *increase* the number of common subexpressions (CS) in a model to increase our benefits from CSE. Third, we extend the approach of flattening *instances* to flattening *problem classes* (Section 5), formally integrate CSE into class-wise flattening, and show when class-wise compilation is preferable to instance-wise compilation. Our empirical analysis (Section 6) confirms our theoretical findings and illustrates the different benefits of instance-wise and class-wise compilation.



**Fig. 1.** Compiling  $n$ -queens problem class (left, top) paired with parameter specification (left bottom), both modelled in language ESSENCE', to input for constraint solver Minion (right).

## 2 Background & Notation

Constraint solvers tackle combinatorial problems that are *modelled* as constraint satisfaction problems (CSP). A CSP consists of a set of *decision variables* and a set of *constraints* on those variables. Each variable represents a choice and is associated with a finite domain of *values* that represent the options for that choice; constraints restrict those options. A solution to a CSP is a value assignment to every variable such that all constraints are satisfied. Typically, constraint problems are modelled as *problem classes* where parameters scale the problem. A *problem instance* (CSP) is obtained by specifying the parameters of a class. Note, that most constraint solvers *only* solve instances.

As an example, consider the  $n$ -queens problem: placing  $n$  queens on a chessboard such that no two queens attack another. The corresponding problem class (Fig. 1, left top), contains  $n$  variables (contained in the array ‘*queens*’), where each variable represents the row-position of a queen and thus ranges over values  $(1..n)$  (line 2). The first constraint (line 4) states that no two queens may be in the same column: the global constraint *alldifferent(queens)* states that every variable in array *queens* has to take a different value. The second and third constraint (line 5,7) disallow two queens positioned on the same NW- and SW-diagonal, respectively. Setting parameter  $n=4$  (like the parameter specification in Fig. 1, left bottom) yields the 4-queens instance.

There exist many popular solver-independent constraint modelling languages, such as OPL [12], MiniZinc [18] or ESSENCE' [7], which provide means to formulate both problem classes and instances. These languages are very expressive and provide high-level constructs to express constraints, such as quantifications. However, most solvers implement only ‘granular’ constraints that allow only variables as arguments<sup>1</sup> (see Fig. 1, right, for 4-queens in solver input language). Therefore, a compilation from modelling language to solver input is necessary, where complex expressions are decomposed into a conjunction of simpler expressions that conform to the granular constraints provided by the solver, a process generally known as *flattening*. Flattening constraints is closely related to flattening expressions in program compilation, with the key difference that *relations* are flattened, instead of *instructions*.

**Definition 1.** We define a **constraint** as an **expression tree**  $E$ , where every node  $N$  in  $E$  corresponds to an operator (*mulop*, *relop* or *Boolean operator*) or *global constraint*;  $N$ 's children are its arguments, and leaves are variables/constants.

<sup>1</sup> Exceptions are solvers such as Eclipse Prolog [4], which flatten their input internally.

---

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

---

**Require:**  $M_S$ : problem instance  
1: **global** *flatCts*, *ctBuffer*, *auxVars*  $\leftarrow$  empty lists  
2: **for all**  $E \in M_S.constraints$  **do**  
3:   *ctBuffer*  $\leftarrow$  empty;  $E'_0 \leftarrow \text{flatten}(E, \text{false})$   
4:    $E'_S \leftarrow E'_0 \wedge (\bigwedge_i E'_i \in \text{ctBuffer}); \text{flatCts.add}(E'_S)$   
5:  $M'_S.constraints \leftarrow \text{flatCts}; M'_S.vars \leftarrow \{M_S.vars \cup \text{auxVars}\}$   
6: **return**  $M'_S$

---

We assume that prior to flattening, every expression tree has been preprocessed such that its tree structure conforms to the granular constraints provided by solver  $S$ . In other words, for every node  $N$  in  $E$ , there exists a constraint 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 provide an  $n$ -ary sum constraint). If all constraints of model  $M$  are adapted to solver  $S$  in such a way, we refer to it as  $M_S$ .

### 3 Flattening Constraint Problem Instances

In this section we formally investigate how to enhance standard instance flattening with common subexpression elimination (CSE) [3, 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.

#### 3.1 A Typical Flattening Approach

Flattening is a well-understood technique, however we summarise a standard flattening algorithm, *flattenInstance*, in Alg. 1 that applies a recursive helper procedure, *flatten* (Alg. 2), on every constraint in instance  $M_S$ . *flatten* iterates over the expression tree in a bottom up fashion and replaces all nodes  $N_i$  in  $E$  (except the root node) with an auxiliary variable  $Aux_i$ , generating the constraint ' $Aux_i = N'_i$ ' that connects every auxiliary variable with its corresponding flat subtree  $N'_i$  (line 7-8 in Alg. 2). After flattening every subnode  $N_i$  of expression  $E$ , the ' $Aux_i = N'_i$ '-constraints are conjoined with the flat root node (line 4 in Alg. 1), yielding the flat representation of  $E$ . As an example, consider Figure 2 that illustrates how *flatten* decomposes the expression  $a \Rightarrow ((x < 3) \wedge (y > 5) \wedge (z = 0))$ . After generating the flat representation  $E'_S$  for every constraint  $E$  in instance  $M_S$ , *flattenInstance* constructs the flat instance  $M'_S$ , consisting of the flat constraints and  $M_S$ 's decision variables combined with the auxiliary variables (line 5) in Alg. 1). Clearly, *flattenInstance* will generate a valid flat instance  $M'_S$ : *flatten*

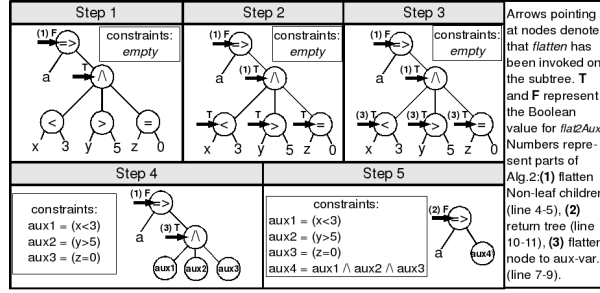
---

**Algorithm 2** *flatten*( $E, \text{flatten2Aux}$ ) recursive procedure that flattens expression tree  $E$

---

**Require:**  $E$ : expression tree, *flatten2Aux*: Boolean  
1: **if**  $\neg(\text{all of } E\text{'s children are leaves})$  **then**  
2:   **for all**  $e_i \in \text{children}(E)$  **do**  
3:     **if**  $\neg(e_i.\text{isLeaf})$  **then**  
4:        $e'_i \leftarrow \text{flatten}(e_i, \text{true})$   
5:        $E.\text{replaceChildWith}(e_i, e'_i)$   
6: **if** *flatten2Aux* **then**  
7:    $Aux \leftarrow \text{createNewVariable}(E.\text{lb}, E.\text{ub}); \text{auxVars.add}(Aux); \text{ctBuffer.add}('Aux = E')$   
8:   **return**  $Aux$   
9: **return**  $E$

---



**Fig. 2.** Example: Flattening ‘ $a \Rightarrow (x < y) \wedge (y > 5) \wedge (z = 0)$ ’ to ‘ $a \Rightarrow aux4 \wedge aux4 \leftrightarrow (aux1 \wedge aux2 \wedge aux3) \wedge aux1 \leftrightarrow (x < 3) \wedge aux2 \leftrightarrow (y > 5) \wedge aux3 \leftrightarrow (z = 0)$ ’

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 3 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 every node, except the root node (*flatten2Aux=false* only for the root node, line 3 in Alg. 1). Thus, *flatten* creates one auxiliary variable and one ‘ $Aux=E$ ’-constraint for all  $m$  nodes, except the  $n$  root nodes. For all  $n$  root nodes, one constraint is returned (line 9 in Alg. 2). Therefore, *flatten* generates  $m - n$  auxiliary variables and  $m$  constraints.  $\square$

**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  $ub$  as synthesised attribute for each node in the expression tree.

### 3.2 Extending Flattening with Common Subexpression Elimination(CSE)

A typical flattening procedure, like *flatten* in Section 3.1, flattens every node in an expression tree to an auxiliary variable. Note, that if two (or more) expression trees have identical subtrees (*common subexpressions*(CS)) 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 CS so as to eliminate these redundancies is desirable.

CSE is a well-established optimisation technique originating from code optimisation [3]. However, unlike in code optimisation, we are dealing with expressions that represent *relations* and not consecutive *instructions*, and thus need not consider the variable’s underlying state. This is similar to expressions in Proof Theory, SAT or Model Checking, where CSE is a wide-spread technique [20, 16, 14]. In most of those disciplines, CSE is performed by transforming expression trees into acyclic directed graphs (DAG) where common nodes are merged [21, 23]. This approach has also been applied to Numerical CSPs [1]. However, CSPs can contain 10,000s of complex constraints, resulting in an extremely large DAG. Therefore, we investigate an alternative approach of CSE, by embedding it into the necessary task of flattening. Note, that exploiting explicit linear equalities, where equivalence is explicitly stated by a constraint, like  $x=y$ , has been well studied in CP [11, 15, 17]. Our work however, is concerned with the advanced case, where equivalence has to be detected.

We formalise CSE-based flattening in the algorithm  $flattenInstance_{CSE}$  (Alg. 3), an extension of standard flattening (Alg. 2).  $flattenInstance_{CSE}$  employs a *hashMap* that maps every flattened subtree 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 Alg. 4 where extensions are highlighted in red font: 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 7), instead 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 7). Clearly,  $flattenInstance_{CSE}$  will flatten each *unique* subnode exactly once.

**Lemma 2.** *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 3 in Alg. 3). If  $E$  has a non-leaf child  $e_i$  (a subnode), then there are two 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  $e_i \rightarrow e'_i$  is added to *hashmap* (line 7); hence, if  $e_i$  appears 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

---

**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** *flatCts*, *ctBuffer*, *auxVars*  $\leftarrow$  empty lists; **global** *hashMap*  $\leftarrow$  empty hashmap  
2: **for all**  $E \in M_S.constraints$  **do**  
3:   *ctBuffer*  $\leftarrow$  empty;  $E'_0 \leftarrow flatten_{CSE}(E, false)$   
4:    $E'_S \leftarrow E'_0 \wedge (\bigwedge_i E'_i \in ctBuffer)$ ; *flatCts.add*( $E'_S$ )  
5:  $M_S.constraints \leftarrow flatCts$ ;  $M'_S.vars \leftarrow \{M_S.vars \cup auxVars\}$   
6: **return**  $M'_S$

---

---

**Algorithm 4** excerpt of *flatten*<sub>CSE</sub>, based on Alg. 2, extensions are given in **red font**.

---

```

Require:  $E$  : expression tree,  $flatten2Aux$  : Boolean flattened to aux var
1: if  $\neg$  (all of  $E$ 's children are leaves) then
2:   for all  $e_i \in children(E)$  do
3:     if  $\neg(e_i.isLeaf)$  then
4:       if  $hashMap.contains(e_i)$  then
5:          $e'_i \leftarrow hashMap.get(e_i)$ 
6:       else
7:          $e'_i \leftarrow flatten_{CSE}(e_i, S, true)$ ;  $hashMap.add(e_i, e'_i)$ 
8:          $E.replaceChildWith(e_i, e'_i)$ 

```

---

root node, and one constraint for every unique root node (see Lemma 1). Therefore,  $M'_S$  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).  $\square$

### 3.3 Comparing Standard Flattening with CSE-based Flattening

We analyse the differences of applying *flattenInstance* and *flattenInstance*<sub>CSE</sub> on problem instance  $M_S$  with  $n$  constraints and  $m$  subexpressions of which  $m_u$  are unique. We denote  $M'$  the flat instance generated by *flattenInstance* and  $M'_{CSE}$  the flat instance generated by *flattenInstance*<sub>CSE</sub>. See Section 6 for an empirical analysis.

**Theorem 1.**  $M'_{CSE}$  contains  $m - m_u$  less constraints and auxiliary variables than  $M'$ .

*Proof.*  $M'$  contains  $m$  constraints and  $m - n$  auxiliary variables (Lemma 1), and  $M'_{CSE}$  contains  $m_u$  constraints and  $m_u - n$  auxiliary variables (Lemma 2). Since  $m \geq m_u$ ,  $M'_{CSE}$  contains  $m - m_u$  less constraints and auxiliary variables than  $M'$ .  $\square$

**Theorem 2.** The space complexity of *flattenInstance* and *flattenInstance*<sub>CSE</sub> is  $O(m)$ .

*Proof.* *flattenInstance* employs the lists *flatCts*, *auxVars* and *ctBuffer*, as well as the representation of constraint model  $M_S$  and  $M'$ . 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*<sub>CSE</sub> 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  $m_u$  unique nodes and the corresponding auxiliary variables, thus the hashmap uses  $2 * m_u * sizeof(String)$  units of memory, where the upper bound of *sizeof(String)* is the size of the longest subexpression in String-format. Therefore, *flattenInstance*<sub>CSE</sub> uses  $2 * m_u * sizeof(String)$  more units of memory than *flattenInstance*. Since  $m_u \leq m$ , the space complexity lies in  $O(m)$ .  $\square$

**Theorem 3.** The time complexity of *flattenInstance* and *flattenInstance*<sub>CSE</sub> is  $O(m)$ .

*Proof.* Let  $f$  be the number of operations 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  $O(m)$ , since  $f \ll m$ . *flattenInstance*<sub>CSE</sub> 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*<sub>CSE</sub> is  $f + 2$  operations. From Lemma

2 we know that  $flattenInstance_{CSE}$  flattens all  $m_u$  unique nodes/subexpression exactly once, hence the overall sum of operations in  $flattenInstance_{CSE}$  is  $m_u*(f+2)$ . In the worst case,  $m_u=m$  (i.e.  $M_S$  has no CS), where  $flattenInstance_{CSE}$  has a runtime of  $m*(f+2)$  where  $f+2 \ll m$  and thus lies in  $O(m)$ .  $\square$

We conclude that there is no significant penalty for attempting CSE during flattening, which we also observe in our experiments (Section 6). Furthermore, if constraint instance  $M_S$  contains common subexpressions, i.e.  $m_u < m$ , the flattening runtime has a less than linear runtime:  $(m-(m-m_u))*(f+2)$  operations are necessary, where  $m-m_u > 0$ . Hence the overall runtime is in  $O(m - c(m))$  where  $c(m) = m - m_u$ . The same holds for space complexity. Consequently, in our experiments (Sec. 6) we observe both space and runtime reductions for instances with many CS, i.e. where  $m_u \ll m$ .

#### 4 Increasing the Number of Common Subexpressions in Instances

In the previous section we have seen that flattening instance  $M_S$  with CSE yields flat constraint instance  $M'_S$  with  $m-m_u$  less constraints and auxiliary variables than standard flattening, reducing the flattening runtime by  $m-m_u$ . Naturally, we are interested in maximising our benefits: if the number of unique nodes  $m_u$  is reduced, (i.e. the number of CS in  $M_S$  increases), instance and runtime reduction,  $m-m_u$ , increases.

The number of unique nodes  $m_u$  in an instance  $M_S$  can be decreased by reformulating equivalent, but not identical subtrees into a common tree structure. For instance, if two subtrees  $E_1$  and  $E_2$  are equivalent but not identical, we can reformulate  $E_2$  to  $E_1$  yielding two identical trees  $E_1$ , which decreases  $m_u$  if performed for every  $E_2$  in the instance. Note, that this reformulation is desirable *only if*  $E_1$  provides *as least as good* propagation as  $E_2$ . Otherwise, reformulating  $E_2$  to  $E_1$  is likely to impair the instance.

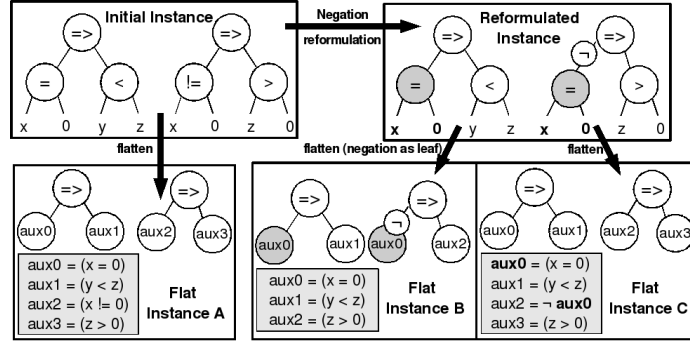
To apply reformulations in a structured, efficient way, a *detection* step is required, where two (or more) equivalent but not identical subtrees are spotted in an instance. Naturally, the effort invested to detect equivalences can be arbitrarily large, especially for very powerful reformulations. For instance, detecting a clique of disequalities to match global constraint *alldifferent* is NP-complete. Therefore, we are interested in measures with low detection effort but high node-reduction potential.

##### 4.1 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 ambiguous representations of such nodes. As an example,  $a \wedge b$  and  $b \wedge a$  are equivalent, but their expression trees are not identical. By imposing an *order* on the arguments of all commutative operators, such ambiguities are eliminated without requiring a detection step. Furthermore, constant *evaluation* can reduce the number of unique nodes by reducing subtrees that consist 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].

##### 4.2 Creating CSs by the Negation Reformulation

The negation reformulation is concerned with reformulating a subtree to its negated form, in order to detect identical subtrees, and was first proposed in [8]. For illustration,



**Fig. 3.** Example for Negation Reformulation: Flattening an instance (top left) with constraints  $(x=0) \Rightarrow (y=z)$  and  $(x \neq 0) \Rightarrow (y > z)$  in three different ways: if flattened directly (with or without CSE), we get *flat instance A* (bottom, left). If applying neg-reformulation on initial instance, we get the reformulated instance (top, right), with 2 common subtrees ‘ $x=0$ ’. If flattened for solvers that allow negated variables as arguments, we get *flat instance B*, otherwise *flat instance C*.

consider a constraint instance containing the two constraints ‘ $(x=0) \Rightarrow (y=z)$ ’ and ‘ $(x \neq 0) \Rightarrow (y > z)$ ’, as depicted in Fig. 3 (top, left). If subtree ‘ $x \neq 0$ ’ is reformulated to ‘ $\neg(x=0)$ ’, the latter contains another ‘ $x=0$ ’, resulting in a CS (see Fig. 3 (top, right)).

Negation can only be extracted from relational or Boolean nodes (e.g. we cannot negate an addition). Since negating Boolean operators typically involves manipulation of the expression tree structure (e.g. de Morgan’s law), we restrict the negation reformulation to relational operators, such as ‘=’ or ‘ $\leq$ ’, where negation corresponds to simply switching operators (e.g. ‘=’ to ‘ $\neq$ ’) and is less expensive to perform.

The detection step is included into flattening of expression tree  $E$  and summarised in Alg. 5 that illustrates the extensions to  $\text{flatten}_{CSE}$  in red font: whenever a non-leaf child  $e_i$  of  $E$  has no CS,  $e_i$  is negated to  $\neg e_i$ , and the hashmap is consulted with  $\neg e_i$  (line 7). If there is a match that returns auxiliary variable  $e'_i$  (line 8),  $e_i$  is replaced by the negated auxiliary variable  $\neg e'_i$  (line 8). If the target solver allows negated variables as constraint arguments (e.g. Minion [6]), then the negated auxiliary variable  $\neg e'_i$  is considered a *leaf* and not further flattened (Fig. 3, *flat instance B*). Otherwise, if the target solver does *not* allow negated variables as arguments in constraints,  $\neg e'_i$  has to be flattened to an auxiliary variable (Fig. 3, *flat instance C*).

As mentioned earlier, it is vital that the reformulation does not impair the model, i.e. the reformulated subtree must provide as least as good propagation as the initial subtree. Therefore, the negation-reformulation may only be applied for solvers where Boolean negation is cheaper than all relational operators it substitutes.

**Theorem 4.** *The time complexity of flattening an instance with  $\text{flatten}_{CSE, neg}$  is  $O(m)$ .*

*Proof.* Say instance  $M_S$  contains  $m_r$  unique relational subnodes (i.e. excluding root nodes), where  $m \geq m_u \geq m_r$ . Then  $\text{flatten}_{CSE, neg}$  will add an additional operation for every subnode (check if node is relational, line 7). Furthermore, for all  $m_r$  subnodes that are relational, it will add  $r$  operations for reformulation (switching the relational operator) and 1 operation for the hashmap check, which is in  $O(1)$ . In summary, it will require  $(m_u - m_r) * (f + 3) + m_r * (f + r + 3)$  operations, which lies in  $O(m)$ , since  $m \geq m_u \geq m_r$  and  $r \ll m$ .  $\square$

---

**Algorithm 5** part of `flattenCSE,neg(E.flatten2Aux)`, a recursive procedure based on Alg. 4, including the negation reformulation for solvers that allow negation of constraint arguments. Extensions are given in **red font**.

---

```

1: if ¬(all of E's children are leaves) then
2:   for all  $e_i \in \text{children}(E)$  do
3:     if ¬( $e_i$ .isLeaf) then
4:       if hashMap.contains( $e_i$ ) then
5:          $e'_i \leftarrow \text{hashMap.get}(e_i)$ 
6:       else
7:         if  $e_i$  is relational & hashMap.contains(¬ $e_i$ ) then
8:            $e'_i \leftarrow \text{hashMap.get}(¬e_i)$ ; hashMap.add( $e_i$ , ¬ $e'_i$ ); E.replaceChildWith( $e_i$ , ¬ $e'_i$ )
9:         else
10:           $e'_i \leftarrow \text{flatten}_{CSE}(e_i, S, \text{true})$ ; hashMap.add( $e_i$ ,  $e'_i$ )
11:          E.replaceChildWith( $e_i$ ,  $e'_i$ )

```

---

## 5 Flattening Problem Classes

In this section, we discuss how to extend flattening *instances* to flattening problem *classes*. The aim of class-wise flattening is to generate the *same* flat instances as instance-wise flattening, but by an alternative approach: first flattening the class and then instantiating parameter values. Other flattening tools, such as the MiniZinc-Flatzinc converter [19], or the internal flattener of Eclipse Prolog [4] are limited to flattening instances, and flattening of constraint problem classes has not been thoroughly investigated. There are two reasons for flattening problem classes: first, to support solvers that take flattened problem classes as input: solvers such as Gecode[5] or Choco [2] are libraries of programming languages, where problems are formulated as programs and parameters can be specified at runtime. Second, it can be more time-efficient to perform flattening *once* at class level instead of *several times*, i.e. once per instance.

In the following, we show how to extend instance-wise flattening to class-wise flattening. First, we extend our notion of expression trees so as to include parameterised expressions. Second, we extend the flattening procedure to deal with parameterised expressions. Finally, we discuss CSE at problem class level.

### 5.1 Representing Parameterised Expressions

Parameters can be used to scale different properties of a constraint class: (1) *constant-scaling* parameters scale constants in constraints, (2) *domain-scaling* parameters scale the domain of decision variables, (3) *variable-scaling* parameters scale the number of variables, and (4) *constraint-scaling* parameters scale the number of constraints.

Constant-scaling parameters appear in leaves of expression trees, where the leaf has no specified, constant domain. Therefore, we extend the node/leaf-attributes *lb* and *ub* (that represent lower and upper bound of every node/leaf) to contain parameters: if parameter *k* appears as a leaf in an expression tree, then its *lb* and *ub* is defined as *k..k*. Similarly, for domain-scaling parameters, 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  $k_1 \leq k_2$ .

Constraint-scaling parameters scale quantifications, such as parameter *n* in the example '**forall**  $i : \text{int}(1..n) . x[i] \neq y[i]$ '. In a problem *instance*, *n* would be specified and the quantification unrolled, generating *n* disequality constraints. However, at class

**Algorithm 6** Excerpt of  $\text{flattenClass}_{CSE}(E, \text{flatten2Aux})$  for flattening problem classes, based on Alg. 4. Extensions are given in **red font**.

---

```

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

```

---

level, such quantifications cannot be unrolled, thus we need to define a notion of quantified expressions as part of our expression tree representation. We extend the expression trees from instance level (Def. 1) 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 in all subnodes/leaves that are quantified. Thus each node/leaf in an expression tree ‘knows’ if (and over what domain) it is quantified (as an example, in Fig. 4 attributes are illustrated as grey labels).

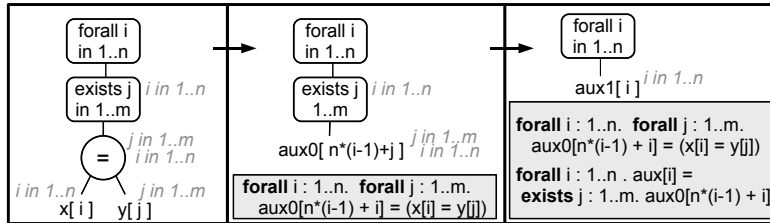
## 5.2 Flattening Quantified Subexpressions

Quantified subexpressions are quantified by parameters, therefore we need to generate a quantified number of auxiliary variables during flattening. Thus, whenever we flatten a quantified subexpression, we generate 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**  $i : \text{int}(1..n) . x[i]=i$ ’. The quantified subexpression ‘ $x[i]=i$ ’ is flattened to a Boolean auxiliary array  $\text{auxArray}$  of length  $n$  (since  $i$  ranges from  $1..n$ ), and ‘ $x[i]=i$ ’ is linked to  $\text{auxArray}$ , resulting in:

$$\text{forall } i : \text{int}(1..n) . \text{auxArray}[i] \Leftrightarrow (x[i] = i)$$

$$\text{exists } i : \text{int}(1..n) . \text{auxArray}[i]$$

We summarise flattening quantified expressions in Alg. 6, which shows an excerpt of the recursive flattening procedure where auxiliary variables and constraints are added to the flat instance; extensions to  $\text{flatten}/\text{flatten}_{CSE}$  are given in **red font**: if node  $E$  is



**Fig. 4. 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.

quantified, it is flattened to an auxiliary variable array (line 3). Note that for convenience, we use 1-dimensional arrays to represent auxiliary variables. The length of the array is determined from the lengths of the quantifying variables' domains: if quantified node  $E$  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. 4, 'aux0' represents an expression that is quantified by  $i \in 1..n$  and  $j \in 1..m$  and hence has length  $n * m$ . Then node  $E$  is linked with *AuxArray* (line 4): if  $E$  is quantified by  $k$  quantifying variables  $v_i$  that range over  $lb_i..ub_i$ , then the linking constraint is:

$$\mathbf{forall} \ v_1 \in (lb_1..ub_1) \dots \mathbf{forall} \ v_k \in (lb_k..ub_k). \\ E = \mathbf{AuxArray}[v_1 + \sum_{2..k}^i (v_i - 1) * \prod_{1..k-1}^j (ub_j - lb_j + 1)]$$

As an example, in Fig. 4, the node 'x[i] = y[j]' which is quantified over  $i \in 1..n$  and  $j \in 1..m$ , is linked to 'aux0' by ' $\forall_{1..n}^i \cdot \forall_{1..m}^j \cdot (x[i] = y[j]) \Leftrightarrow \mathbf{aux0}[i + (j - 1) * n]$ '.

**Theorem 5.** *The time complexity of flattening with flattenClass is in  $O(m)$ .*

*Proof.* *flattenClass* (Alg. 6) is an extension of both Alg. 2 and Alg. 4 which additionally flattens quantified expressions. Say  $f$  is the number of operations necessary for flattening unquantified nodes, and  $f_q$  the number for flattening quantified nodes. From Theorem 3 we know that the flattening procedure flattens every node exactly once. Therefore, if class  $M_s$  contains  $m_q$  quantified nodes, then *flattenClass* will take  $m_q * f_q + (m - m_q) * f$  operations, which lies in  $O(m)$  since  $f_q \ll m_q$ .  $\square$

**CSE at Problem Class Level.** CSE as described for problem instances can be applied to problem classes. As an example for CSE on class level, consider the NW-diagonal constraint from  $n$ -queens (Fig. 1):

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

After flattening 'queens[i] + i' to array 'aux0[i]', the flattening engine flattens 'queens[j] + j', which matches the former, since  $j$  ranges over the same domain as  $i$ . Thus 'queens[j] + j' is also represented by 'aux0' and the constraint flattened to:

$$\forall i : \text{int}(1..n). \ \mathbf{aux0}[i] = \text{queens}[i] + i \\ \forall i, j : \text{int}(1..n). \ (i \neq j) \Rightarrow (\mathbf{aux0}[i] \neq \mathbf{aux0}[j])$$

However, CSE at the class level must be done with care: consider, for example, two quantified 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. Furthermore, two quantified nodes 'x[i]\*y[i]' and 'x[j]\*y[j]' are equivalent if quantifying variables  $i$  and  $j$  range over the same domain. We currently represent quantified variables by their respective domain when they are entered into the hashmap for matching CSs. For instance, if  $i$  ranges from  $1..n$  in node 'x[i]\*y[i]' the node is stored as 'x[{1..n}]\*y[{1..n}]' in the hashmap and hence only matched with expressions that are quantified over the exactly same domain. Note however, that we also need to compare the number of quantifying variables of each node/leaf, since otherwise nodes like 'x[i]\*y[i]' and 'x[i]\*y[j]' would be considered equivalent.

### 5.3 Current Limitations & Future Work

**Redundancies.** Quantified expressions can be guarded by expressions, like ' $(i \neq j)$ ' in:

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

Flattening will introduce  $2 * n^2$  auxiliary variables (one array of length  $n^2$  for each multiplication), however, since  $(i \neq j)$  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 during instance-wise flattening, where constant evaluation reduces expressions like ‘ $false \Rightarrow E$ ’ to  $true$ , 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 best to 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).

**Undetected Common Subexpressions.** Our current approach for detecting CSs at the class level does not detect all CSs that instance-wise flattening detects. For instance, consider the constraint ‘ $\forall_{1..n-1} . x[i]\%n + x[i+1]\%n = 1$ ’ which the current flattening approach flattens to ‘ $\forall_{1..n-1} . aux0[i] + aux1[i+1] = 1$ ’, using ‘aux0’ for ‘ $x[i]\%n$ ’ and ‘aux1’ for ‘ $x[i+1]\%n$ ’, since both nodes are quantified over different domains. However, ‘ $x[i]\%n$ ’ and ‘ $x[i+1]\%n$ ’ have ‘shifted’ CSs for  $n > 2$ , hence the same auxiliary array can be used:

$$\begin{aligned} \forall i : \text{int}(1..n) . \mathbf{aux}[i] &= x[i]\%n \\ \forall i : \text{int}(1..n-1) . \mathbf{aux}[i] + \mathbf{aux}[i+1] &= 1 \end{aligned}$$

To detect this equivalence one has to reason about the quantifying domains of both nodes:  $x[i]\%n$  has domain  $D_1 = \{1..n-1\}$  and  $x[i+1]\%n$  has  $D_2 = \{2..n\}$  where  $D_1 \cap D_2 \neq \emptyset$ , so one can replace both expressions with the same auxiliary variable that ranges over  $D_1 \cup D_2$ , i.e.  $\{1..n\}$ . The CSE-hashmap cannot perform this sort of reasoning. In the future we will investigate how to optimise CS detection at the class level. Nevertheless, we report below promising empirical results with our limited technique.

#### 5.4 Instance-wise versus Class-wise Compilation

Typical CSP compilation proceeds *instance-wise*: a class is first merged with a parameter specification, yielding an unflattened instance which is then flattened to a flat instance (Fig. 5, left). We propose *class-wise* compilation, an alternative approach, where first the problem class is flattened to a flat class, which is then merged with a parameter specification, yielding a flat instance (Fig. 5, right). Note, that the flat instances from instance- and class-wise compilation are identical, with certain exceptions (see Sec. 5.3). We investigate the circumstances under which class-wise compilation is preferable to instance-wise compilation, and vice versa.

Say we want to compile  $k$  instances of problem class  $C$ . Instance-wise compilation merges  $C$  with a parameter specification and then flattens it,  $k$  times respectively. Class-wise compilation flattens  $C$  once and then merges the flat class  $k$  times with a

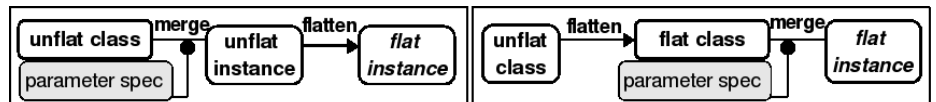


Fig. 5. instance-wise compilation (left) and class-wise compilation (right)

parameter specification. From Theorem 3 and 5 we know that both instance-wise and class-wise flattening lie in  $O(m)$ , where  $m$  is the number of nodes in the instance/class. Let  $merge(m, i)$  be the time complexity of merging a problem class with  $m$  nodes with parameter specification  $i$  (which includes unrolling quantifications, normalisation, etc). Let  $m_i$  be the number of nodes of unflat instance  $i$ ,  $m$  the number of nodes in unflattened class  $C$ , and  $m'$  the number of nodes in flat class  $C$ , with  $m \leq m'$  and  $m \leq m_i$ . The time complexity of instance- and class-wise compilation of  $k$  instances is:

$$\sum_{1..k}^i (O(m_i) + merge(m, i)) \quad \text{Runtime for *instance-wise* compilation}$$

$$O(m) + \sum_{1..k}^i merge(m', i) \quad \text{Runtime for *class-wise* compilation}$$

The preferred compilation process for class  $C$  depends on  $m \sim m'$  and the number of instances  $k$ . If  $C$  has no constraint-scaling parameters, every instance will have the same number of nodes as the flat class, i.e.  $m' = m_1 = \dots = m_k$ . Thus, class-wise compilation lies in  $O(m) + k * merge(m', i)$  and instance-wise compilation in  $k * O(m') + k * merge(m, i)$ . For small  $k$  and classes where  $m \ll m'$ , instance-wise compilation takes less runtime, otherwise class-wise compilation is faster. Otherwise, if problem class  $C$  contains constraint-scaling parameters,  $m' \leq m_i$ , i.e. the number of nodes in flat  $C$  will be less than or equal to the number of nodes in every unflat instance. If there are many instances with  $m' \leq m_i$  and  $k$  is small, instance-wise compilation will be faster, otherwise class-wise compilation is to be preferred. For an empirical analysis see Tab. 1.

## 6 Experimental Results

In our empirical evaluation we first investigate instance-wise flattening, where we compare standard flattening, flattening with CSE, and the negation reformulation. Second, we compare instance-wise with class-wise compilation. We use a selection of problems that have different numbers of CSs. Each problem is formulated in ESSENCE' [7] compiled to two different target solvers, Minion [6] and Gecode [5], using the compiler TAILOR [7], that performs CSE and negation reformulation automatically at 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.

**Instance-wise Flattening.** Fig. 6 (1) depicts the instance reduction with CSE-based flattening and the negation reformulation. We see that in some cases the number of unique nodes  $m_u$  is only  $\frac{1}{10}$  of the number of all nodes  $m$ , hence CSE-based flattening reduces the number of constraints/auxiliary variables to  $\frac{1}{10}$  compared to standard flattening. Furthermore, in Fig. 6(2) we can see that the flattening time for most instances

Problem	Compilation Time (sec)						$m$	$m'$	$m'_i$
	Class	Inst1	Inst2	Inst3	Inst4	Sum			
Peg Solitaire (class)	17.96	6.245	6.235	6.237	6.240	<b>42.917</b>	30	5425	5425
(instance)	-	17.247	17.227	17.230	17.345	59.049			
BIBD (class)	0.168	0.335	4.679	9.653	16.875	<b>31.710</b>	17	20	$2(b+v-1) + (b+1)(v^2+v)$
(instance)	-	0.306	8.774	19.044	33.222	61.346			
Langford (class)	0.155	0.196	0.197	0.195	0.197	0.940	5	5	$k*(l-1)+1$
(instance)	-	0.196	0.197	0.194	0.197	<b>0.784</b>			

**Table 1.** Class-wise vs. Instance-wise Compilation with  $k=4$  instances. (class): class-wise flattening: first flatten class, then merge with  $Inst_i$ . (instance): instance-wise flattening: merge+flatten every  $Inst_i$ .  $m$ : #nodes in unflat class,  $m'$ : #nodes in flat class,  $m'_i$ : #nodes in flat instance  $i$

is *reduced* during CSE-based flattening, hence in many cases, we get smaller instances for less effort in time. When comparing memory usage (Fig. 6 (3)) we observe that we use less than 40 % memory with CSE-based flattening. Note also, that for problems that have no CSs ( $m=m_u$ ), such as BIBD or the Golomb Ruler (Gecode), we pay no significant overhead for attempting CSE. Most importantly, we observe impressive speed-ups in solving time (up to 3,000) due to CSE/negation reformulation (Fig. 6 (4)) (for reasons of space we only show speedups for Minion).

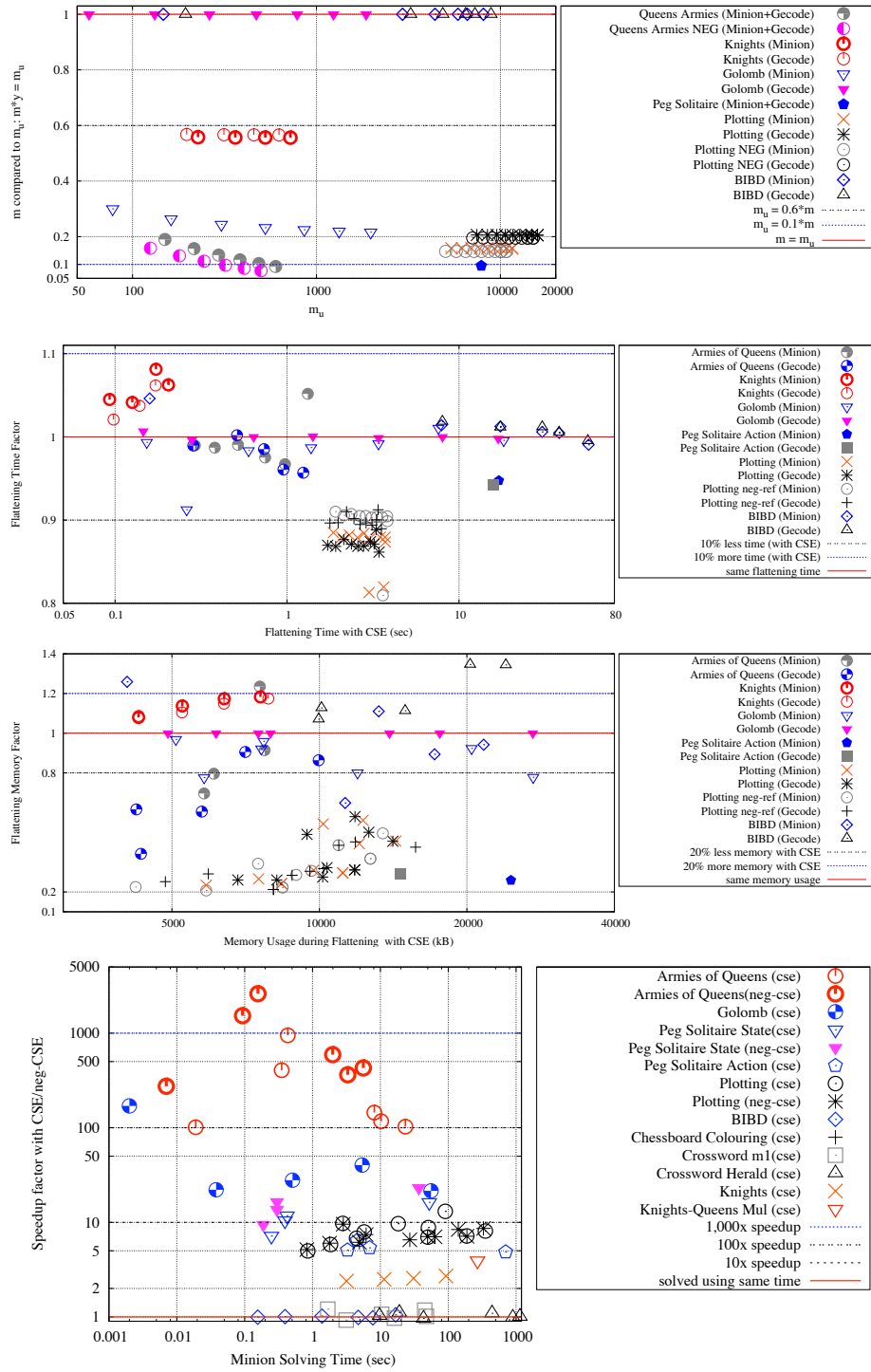
**Class-wise versus Instance-wise Compilation.** We compare class-wise with instance-wise compilation on a small selection of problems with different characteristics: (1) Peg Solitaire has no constraint-scaling parameters, in (2) BIBD,  $m < m'$  and (3) Langford where  $m = m'$ . We apply both instance- and class-wise compilation on 4 instances and compare the overall compilation time in Tab. 1: class-wise compilation is faster with (1) and (2) and competitive with (3).

## 7 Conclusion

In this paper, we have given a *theoretical and empirical analysis* of the benefits of integrating common subexpression elimination (CSE) into the necessary process of flattening. We show how to *increase* the number of common subexpressions by reformulation for little computational effort. Furthermore, we augment flattening to problem *class level*, show how to perform CSE and present an alternative compilation approach of *class-wise* flattening, which shows to be a promising alternative to standard instance-wise compilation at theoretical and practical level.

## 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. Cocks, 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. P.J. Stuckey, R. Becket, S. Brand, M. Brown, T. Feydy, J. Fischer, M. Garcia de la Banda, K. Marriot and M. Wallace The Evolving World of MiniZinc in *ModRef* 2009, pp.156-169, 2009.
20. D.A. Plaisted, and S. Greenbaum A structure-preserving clause form translation, *Jnl of Computation* 2,293-304
21. H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization *Journal of Global Optimization* 33/4 (2005), 541-562
22. 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.
23. 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



**Fig. 6. Instance Reduction (1).** The  $x$ -axis represents the number of unique nodes  $m_u$ , the  $y$ -axis shows the reduction factor compared to  $m$ , hence points below  $y=1$  depict the cases when  $m_u < m$ . 'NEG' denotes instances generated with the negation reformulation. **Flattening Time (2)** and **Memory Usage (3)** during flattening. The  $x$ -axis represents the time/memory for flattening with CSE, the  $y$ -axis shows the factor for comparison with standard flattening: points below  $y=1$  depict cases where flattening time/space was reduced with CSE, points above when increased. **Speedup in Minion (4)**. The  $x$ -axis represents the solving time (sec) with CSE/negation reformulation (NEG), the  $y$ -axis shows the speedup factor, hence points above  $y=1$  depict the cases when the instance was solved faster with CSE/NEG.